Linux内核完全注释

内核版本0.11(0.95)

赵　炯　著



www.plinux.org　　gohigh@sh163.net

Linux 内核 0.11 完全注释

A Heavy Commented Linux Kernel Source Code

Linux Version 0.11

（Revision 1.1）

Zhao Jiong

赵炯

gohigh@sh163.net

www.plinux.org

www.oldlinux.org

2003-10-6

内容简介

本书对 Linux 早期操作系统内核(v0.95)全部代码文件进行了详细全面的注释和说明，旨在使读者能够在尽量短的时间内对 Linux 的工作机理获得全面而深刻的理解，为进一步学习和研究 Linux 系统打下坚实的基础。虽然所选择的版本较低，但该内核已能够正常编译运行，其中已经包括了 LINUX 工作原理的精髓，通过阅读其源代码能快速地完全理解内核的运作机制。书中首先以 Linux 源代码版本的变迁历史为主线，详细介绍了 Linux 系统的发展历史，着重说明了各个内核版本之间的重要区别和改进方面，给出了选择 0.11(0.95)版作为研究的对象的原因。另外介绍了内核源代码的组织结构及相互关系，同时还说明了编译和运行该版本内核的方法。然后本书依据内核源代码的组织结构对所有内核程序和文件进行了注释和详细说明。每章的安排基本上分为具体研究对象的概述、每个文件的功能介绍、代码内注释、代码中难点及相关资料介绍、与当前版本的主要区别等部分。最后一章内容总结性地介绍了继续研究 Linux 系统的方法和着手点。

版权说明

作者保留本电子书籍的修改和正式出版的所有权利.读者可以自由传播本书全部和部分章节的内容,但需要注明出处.由于目前本书尚为草稿阶段,因此存在许多错误和不足之处,希望读者能踊跃给予批评指正或建议.可以通过电子邮件给我发信息:gohigh@sh163.net, 或直接来信至:上海同济大学 机械电子工程研究所(上海四平路 1239 号,邮编:200092).

© 2002,2003 by Zhao Jiong

© 2002,2003 赵炯 版权所有.

"RTFSC – Read The F**king Source Code ☺!"

–Linus Benedict Torvalds

目录

文件列表索引

序言

本书是一本有关 Linux 操作系统内核基本工作原理的入门读物。

本书的主要目标

本书的主要目标是使用尽量少的篇幅或在有限的篇幅内，对完整的 Linux 内核源代码进行解剖，以期对操作系统的基本功能和实际实现方式获得全方位的理解。做到对 linux 内核有一个完整而深刻的理解，对 linux 操作系统的基本工作原理真正理解和入门。

本书读者群的定位是一些知晓 Linux 系统的一般使用方法或具有一定的编程基础，但比较缺乏阅读目前最新内核源代码的基础知识,又急切希望能够进一步理解 UNIX 类操作系统内核工作原理和实际代码实现的爱好者。这部分读者的水平应该界于初级与中级水平之间。目前，这部分读者人数在 Linux 爱好者中所占的比例是很高的，而面向这部分读者以比较易懂和有效的手段讲解内核的书籍资料不多。

现有书籍不足之处

目前已有的描述 Linux 内核的书籍,均尽量选用最新 Linux 内核版本(例如 Redhat 7.0 使用的 2.2.16 稳定版等）进行描述，但由于目前 Linux 内核整个源代码的大小已经非常得大（例如 2.2.20 版具有 268 万行代码!)，因此这些书籍仅能对 Linux 内核源代码进行选择性地或原理性地说明，许多系统实现细节被忽略。因此并不能给予读者对实际 Linux 内核有清晰而完整的理解。

陆丽娜等同志翻译的 Scott Maxwell 著的一书《Linux 内核源代码分析》基本上是面对 Linux 中级水平的读者，需要较为全面的基础知识才能完全理解。而且可能是由于篇幅所限，该书并没有对所有 Linux 内核代码进行注释，略去了很多内核实现细节，例如其中内核中使用的各个头文件(*.h)、生成内核代码映象文件的工具程序、各个 make 文件的作用和实现等均没有涉及。因此对于处于初中级水平之间的读者来说阅读该书有些困难。

去年初浙江大学出版的《Linux 内核源代码情景分析》一书，也基本有这些不足之处。甚至对于一些具有较高 Linux 系统应用水平的计算机本科高年级学生，由于该书篇幅问题以及仅仅选择性地讲解内核源代码，也不能真正吃透内核的实际实现方式，因而往往刚开始阅读就放弃了。这在本人教学的学生中基本都会出现这个问题。该书刚面市时，本人曾极力劝说学生购之阅读，并在二个月后调查阅读学习情况，基本都存在看不下去或不能理解等问题，大多数人都放弃了。

John Lions 著的《莱昂氏 UNIX 源代码分析》一书虽然是一本学习 UNIX 类操作系统内核源代码很好的书籍，但是由于其采用的是 UNIX V6 版，其中系统调用等部分代码是用早已过时的 PDP-11 系列机的汇编语言编制的，因此在阅读与硬件部分相关的源代码时就会遇到较大的困难。

A.S.Tanenbaum 的书《操作系统：设计与实现》是一本有关操作系统内核实现很好的入门书籍，但该书所叙述的 minix 系统是一种基于消息传递的内核实现机制，与 Linux 内核的实现有所区别。因此在学习该书之后，并不能很顺利地即刻着手进一步学习较新的 Linux 内核源代码实现。

在使用这些书籍进行学习时会有一种"盲人摸象"的感觉，不能真正理解 Linux 内核系统具体实现的整体概念，尤其是对那些 Linux 系统初学者或刚学会如何使用 Linux 系统的人在使用那些书学习内核原理时，内核的整体运作结构并不能清晰地在脑海中形成。这在本人多年的 Linux 内核学习过程中也深

有体会，也是写作本书的动机之一。

为了填补这个空缺，本书的主要目标是使用尽量少的篇幅或在有限的篇幅内，对完整的 Linux 内核源代码进行全面解剖，以期对操作系统的基本功能和实际实现方式获得全方位的理解。做到对 linux 内核有一个完整而深刻的理解，对 linux 操作系统的基本工作原理真正理解和入门。

阅读早期内核其它的好处？

目前，已经出现不少基于 Linux 早期内核而开发的专门用于嵌入式系统的内核版本，如 DJJ 的 x86 操作系统、Uclinux 等（在 www.linux.org 上有专门目录），世界上也有许多人认识到通过早期 Linux 内核源代码学习的好处，目前国内也已经有人正在组织人力注释出版类似本文的书籍。因此，通过阅读 Linux 早期内核版本的源代码，的确是学习 Linux 系统的一种行之有效的途径，并且对研究和应用 Linux 嵌入式系统也有很大的帮助。

在对早期内核源代码的注释过程中，作者已经发现，早期内核源代码几乎就是目前所使用的较新内核的一个精简版本。其中已经包括了目前新版本中几乎所有的基本功能原理的内容。正如《系统软件：系统编程导论》一书的作者 Leland L. Bach 在介绍系统程序以及操作系统设计时，引入了一种极其简化的简单指令计算机(SIC)系统来说明所有系统程序的设计和实现原理，从而既避免了实际计算机系统的复杂性，又能透彻地说明问题。这里选择 Linux 的早期内核版本作为学习对象，其指导思想与 Leland 的一致。这对 Linux 内核学习的入门者来说，无非是一个最理想的选择之一。能够在尽可能短的时间内深入理解 Linux 内核的基本工作原理。

当然，使用早期内核作为学习的对象也有不足之处，所选用的 Linux 早期内核版本不包含对虚拟文件系统 VFS 的支持、对网络系统的支持、仅支持 a.out 执行文件和对其它一些现有内核中复杂子系统的说明。但由于本书是作为 Linux 内核工作机理实现的入门教材，因此这也正是选择早期内核版本的优点之一。通过学习本书，可以为进一步学习这些高级内容打下扎实的基础。

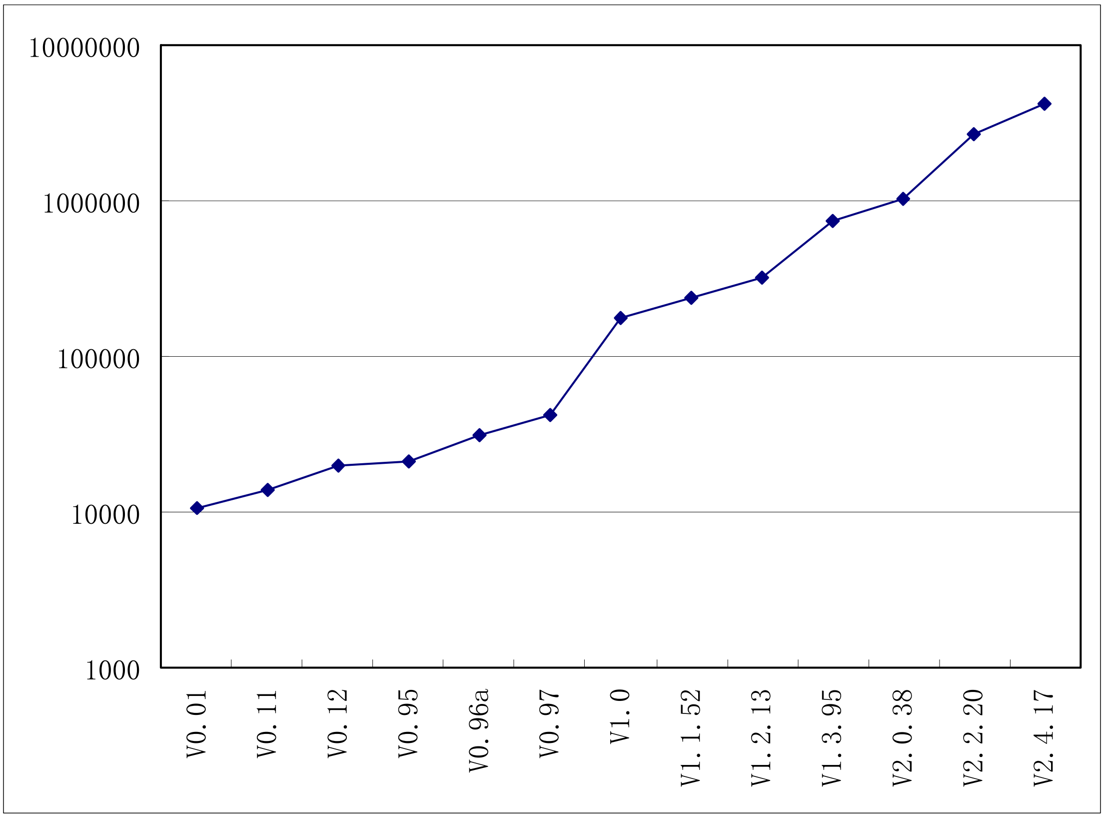
阅读完整源代码的重要性和必要性

正如 Linux 系统的创始人在一篇新闻组投稿上所说的，要理解一个软件系统的真正运行机制，一定要阅读其源代码（RTFSC – Read The Fucking Source Code）。系统本身是一个完整的整体，具有很多看似不重要的细节存在，但是若忽略这些细节，就会对整个系统的理解带来困难，并且不能真正了解一个实际系统的实现方法和手段。

虽然通过阅读一些操作系统原理经典书籍（例如 M.J.Bach 的《UNIX 操作系统设计》）能够对 UNIX 类操作系统的工作原理有一些理论上的指导作用，但实际上对操作系统的真正组成和内部关系实现的理解仍不是很清晰。正如 AST 所说的，“许多操作系统教材都是重理论而轻实践”，“多数书籍和课程为调度算法耗费大量的时间和篇幅而完全忽略 I/O，其实，前者通常不足一页代码，而后者往往要占到整个系统三分之一的代码总量。”内核中大量的重要细节均未提到。因此并不能让读者理解一个真正的操作系统实现的奥妙所在。只有在详细阅读过完整的内核源代码之后，才会对系统有一种豁然开朗的感觉，对整个系统的运作过程有深刻的理解。以后再选择最新的或较新内核源代码进行学习时，也不会碰到大问题，基本上都能顺利地理解新代码的内容。

如何选择要阅读的内核代码版本

那么，如何选择既能达到上述要求，又不被太多的内容而搞乱头脑，选择一个适合的 Linux 内核版本进行学习，提高学习的效率呢？作者通过对大量内核版本进行比较和选择后，最终选择了与目前 Linux 内核基本功能较为相近，又非常短小的 0.11 版内核作为入门学习的最佳版本。下图是对一些主要 Linux 内核版本行数的统计。



目前的 Linux 内核源代码量都在几百万行的数量上，极其庞大，对这些版本进行完全注释和说明是不可能的，而 0.11 版内核不超过 2 万行代码量，因此完全可以在一本书中解释和注释清楚。麻雀虽小，五脏俱全。

另外，使用该版本可以避免使用现有较新内核版本中已经变得越来越复杂得各子系统部分的研究（如虚拟文件系统 VFS、ext2 或 ext3 文件系统、网络子系统、新的复杂的内存管理机制等）。

阅读本书需具备的基础知识

在阅读本书时，作者希望读者具有以下一些基础知识或有相关的参考书籍在手边。其一是有关 80x86 处理器结构和编程的知识或资料。例如可以从网上下载的 80x86 编程手册（INTEL 80386 Programmer's Reference Manual）；其二是有关 80x86 硬件体系结构和接口编程的知识或资料。有关这方面的资料很多；其三还应具备初级使用 Linux 系统的简单技能。

另外，由于 Linux 系统内核实现，最早是根据 M.J.Bach 的《UNIX 操作系统设计》一书的基本原理开发的，源代码中许多变量或函数的名称都来自该书，因此在阅读本书时，若能适当参考该书，更易于对源代码的理解。

Linus 在最初开发 Linux 操作系统时，参照了 Minix 操作系统。例如，最初的 Linux 内核版本完全照搬了 Minix 1.0 文件系统。因此，在阅读本书时，A.S.Tanenbaum 的书《操作系统：设计与实现》也具有较大的参考价值。但 Tanenbaum 的书描述的是一种基于消息传递在内核各模块之间进行通信（信息交换），这与 Linux 内核的工作机制不一样。因此可以仅参考其中有关一般操作系统工作原理章节和文件系统实现的内容。

使用早期版本是否过时？

表面看来本书对 Linux 早期内核版本注释的内容犹如 Linux 操作系统刚公布时 Tanenbaum 就认为其已经过时的（Linux is obsolete）想法一样，但通过学习本书内容，你就会发现，利用本书学习 Linux 内核，由于内核源代码量短小而精干，因此会有极高的学习效率，能够做到事半功倍，快速入门。并且对继续进一步选择新内核部分源代码的学习打下坚实的基础。在学习完本书之后，你将对系统的运作原理有一个非常完整而实际的概念，这种完整概念能使人很容易地进一步选择和学习新内核源代码中的任何部分，而不需要再去啃读代码量巨大的新内核中的完整的源代码。

Ext2 文件系统与 Minix 文件系统？

目前 Linux 系统上所使用的 Ext2（或最新的 Ext3）文件系统，是在内核 1.x 之后开发的文件系统，其功能详尽并且其性能也非常完整和稳固，是目前 Linux 操作系统上默认的标准文件系统。但是，作为对 Linux 操作系统完整工作原理入门学习所使用的部分，原则上是越精简越好。为了达到对一个操作系统有完整的理解，并且能不被其中各子系统中的复杂性和过多的细节喧宾夺主，在选择学习剖析用的内核版本时，只要系统的部分代码内容能说明实际工作原理，就越简单越好。

Linux 内核 0.11 版上当时仅包含最为简单的 minix 1.0 文件系统，对于理解一个操作系统中文件系统的实际组成和工作原理已经足够。这也是选择 Linux 早期内核版本进行学习的主要原因之一。

在完整阅读完本书之后，相信您定会发出这样的感叹：“对于 Linux 内核系统，我现在终于入门了！”。此时，您应该有十分的把握去进一步学习最新内核中各部分的工作原理和过程了。

同济大学
赵炯 博士
2002.12

第 1 章 概述

本章首先回顾了 Linux 操作系统的诞生、开发和成长过程，由此理解本书为什么会选择 Linux 系统早期版本作为学习对象的一些原因。然后具体说明了选择早期 Linux 内核版本进行学习的优点和不足之处以及如何开始进一步的学习。

1.1 Linux 的诞生和发展

Linux 操作系统是 UNIX 操作系统的一种克隆系统。它诞生于 1991 年的 10 月 5 日（这是第一次正式向外公布的时间）。以后借助于 Internet 网络，并经过全世界各地计算机爱好者的共同努力下，现已成为今天世界上使用最多的一种 UNIX 类操作系统，并且使用人数还在迅猛增长。

Linux 操作系统的诞生、发展和成长过程始终依赖着以下五个重要支柱：UNIX 操作系统、MINIX 操作系统、GNU 计划、POSIX 标准和 Internet 网络。

下面主要根据这五个基本线索来追寻一下 Linux 的开发历程，它的酝酿过程，最初的发展经历。首先分别介绍其中的四个基本要素(UNIX、MINIX、GNU 和 POSIX，Internet 的重要性显而易见，所以不用对其罗嗦)，然后根据 Linux 的创始人 Linus Toravlds 从对计算机感兴趣而自学计算机知识，到心里开始酝酿编制一个自己的操作系统，到最初 Linux 内核 0.01 版公布，以及从此如何艰难地一步一个脚印地在全世界 hacker 的帮助下最后推出比较完善的 1.0 版本这段时间的发展经过,也即对 Linux 的早期发展历史进行详细介绍。

当然，目前 Linux 内核版本已经开发到了 2.5.52 版。而大多数 Linux 系统中所用到的内核是稳定的 2.4.20 版内核。（其中第 2 个数字奇数表示是正在开发的版本，不能保证系统的稳定性）对于 Linux 的一般发展史，许多文章和书籍都有介绍，这里就不重复。

1.1.1 UNIX 操作系统的诞生

Linux 操作系统是 UNIX 操作系统的一个克隆版本。UNIX 操作系统是美国贝尔实验室的 Ken.Thompson 和 Dennis Ritchie 于 1969 年夏在 DEC PDP-7 小型计算机上开发的一个分时操作系统。

当时 Ken Thompson 为了能在闲置不用的 PDP-7 计算机上运行他非常喜欢的星际旅行(Space travel）游戏，在 1969 年夏天乘他夫人回家乡加利福尼亚渡假期间，在一个月内开发出了 unix 操作系统的原型。当时使用的是 BCPL 语言（基本组合编程语言），后经 Dennis Ritchie 于 1972 年用移植性很强的 C 语言进行了改写，使得 UNIX 系统在大专院校得到了推广。

1.1.2 MINIX 操作系统

MINIX 系统是由 Andrew S. Tanenbaum（AST）开发的。AST 是在荷兰 Amsterdam 的 Vrije 大学数学与计算机科学系统工作，是 ACM 和 IEEE 的资深会员(全世界也只有很少人是两会的资深会员)。共发表了 100 多篇文章，5 本计算机书籍。

AST 虽出生在美国纽约，但是是荷兰侨民(1914 年他的祖辈来到美国)。他在纽约上的中学、M.I.T 上的大学、加洲大学 Berkeley 分校念的博士学位。由于读博士后的缘故，他来到了家乡荷兰。从此就与家乡一直有来往。后来就在 *Vrije* 大学开始教书、带研究生了。荷兰首都 Amsterdam 是个常年阴雨绵绵的城市，而对于 AST 来说，这最好不过了，因为这样他就可以待在家里摆弄他的计算机了。

MINIX 是他 1987 年编制的，主要用于学生学习操作系统原理。到 91 年时版本是 1.5。目前主要有两个版本在使用： 1.5 版和 2.0 版，当时该操作系统在大学使用是免费的，但其它用途不是，当然目前都

已经是免费的，可以从许多 FTP 上下载。

对于 Linux 系统，他表示对其开发者 Linus 的称赞。但他认为 Linux 的发展有很大原因是因为他为了保持 minix 的小型化，能让学生在一个学期内就能学完，而没有接纳全世界许多人对 Minix 的扩展要求。因此这激发了 Linus 编写 Linux。Linus 正好抓住了这个好时机。

作为一个操作系统，MINIX 并不是优秀者，但它同时提供了用 C 语言和汇编语言写的系统源代码。这是第一次使得有抱负的程序员或 hacker 能够阅读操作系统的源代码,在当时这种源代码是软件商一直小心地守护着的。

1.1.3 GNU 计划

GNU 计划和自由软件基金会(the Free Software Foundation - FSF)是由 Richard M. Stallman 于 1984 年一手创办的。旨在开发一个类似 Unix、并且是自由软件的完整操作系统:GNU 系统。(GNU 是"GNU's Not Unix"的递归缩写，它的发音为"guh-NEW"。) 各种使用 linux 作为核心的 GNU 操作系统正在被广泛的使用。虽然这些系统通常被称作"Linux"，但是严格地说，它们应该被称为 GNU/Linux 系统。

到上世纪 90 年代初,GNU 项目已经开发出许多高质量的免费软件,其中包括有名的 emacs 编辑系统、bash shell 程序、gcc 系列编译程序、gdb 调试程序等等。这些软件为 Linux 操作系统的开发创造了一个合适的环境,是 Linux 能够诞生的基础之一。以至于目前许多人都将 Linux 操作系统称为“GNU/Linux”操作系统。

1.1.4 POSIX 标准

POSIX(Portable Operating System Interface for Computing Systems)是由 IEEE 和 ISO/IEC 开发的一簇标准。该标准是基于现有的 UNIX 实践和经验，描述了操作系统的调用服务接口，用于保证编制的应用程序可以在源代码一级上在多种操作系统上移植运行。它是在 1980 年早期一个 UNIX 用户组(usr/group)的早期工作的基础上取得的。该 UNIX 用户组原来试图将 AT&T 的系统 V 和 Berkeley CSRG 的 BSD 系统的调用接口之间的区别重新调和集成,从而于 1984 年产生了/usr/group 标准。1985 年,IEEE 操作系统技术委员会标准小组委员会（TCOS-SS）开始在 ANSI 的支持下责成 IEEE 标准委员会制定有关程序源代码可移植性操作系统服务接口正式标准。到了 1986 年 4 月，IEEE 就制定出了试用标准。第一个正式标准是在 1988 年 9 月份批准的（IEEE 1003.1-1988），也既以后经常提到的 POSIX.1 标准。

1989 年 POSIX 的工作被转移至 ISO/IEC 社团,并由 15 工作组继续将其制定成 ISO 标准。到 1990 年,POSIX.1 与已经通过的 C 语言标准联合，正式批准为 IEEE 1003.1-1990（也是 ANSI 标准）和 ISO/IEC 9945-1:1990 标准。

POSIX.1 仅规定了系统服务应用程序编程接口（API），仅概括了基本的系统服务标准，因此期望对系统的其它功能也制定出标准。这样 IEEE POSIX 的工作就开始展开了。在 1990 年，刚开始有十个批准的计划在进行，有近 300 多人参加每季度为期一周的会议。着手的工作有命令与工具标准(POSIX.2)、测试方法标准（POSIX.3）、实时 API（POSIX.4）等。到了 1990 年上半年已经有 25 个计划在进行，并且有 16 个工作组参与了进来。与此同时，还有一些组织也在制定类似的标准，如 X/Open，AT&T，OSF 等。

在 90 年代初,POSIX 标准的制定正处在最后投票敲定的时候，那是 1991-1993 年间。此时正是 Linux 刚刚起步的时候,这个 UNIX 标准为 Linux 提供了极为重要的信息,使得 Linux 的能够在标准的指导下进行开发,能够与绝大多数 UNIX 系统兼容。在最初的 Linux 内核代码中(0.01 版、0.11 版)就已经为 Linux 与 POSIX 标准的兼容做好了准备工作。在 0.01 版的内核/include/unistd.h 文件中就已经定义了几个有关 POSXI 标准要求的常数符号，并且在注释中就写到“ok，这也许是个玩笑，但我正在着手研究它呢”。

1991 年 7 月 3 日在 comp.os.minix 上发布的 post 上就已经提到了正在搜集 POSIX 的资料。(当然此时还不存在 Linux 这个名称，当时 Linus 的脑子里想的可能是 FREAX ☺，FREAX 的英文含义是怪诞的、怪物、异想天开等)。其中透露了他正在进行 Linux 系统的开发，并且在 Linux 最初的时候已经想到要实现与 POSIX（UNIX 的国际标准）的兼容问题了。

1.1.5 Linux 操作系统的诞生

1981 年 IBM 公司推出享誉全球的微型计算机 IBM PC。在 1981-1991 年间，MS-DOS 操作系统一直是微型计算机上操作系统的主宰。此时计算机硬件价格虽然逐年下降，但软件价格仍然是居高不下。当时 Apple 的 MACs 操作系统可以说是性能最好的，但是其天价没人能够轻易靠近。

当时的另一个计算机技术阵营是 Unix 世界。但是 Unix 操作系统就不仅是价格贵的问题了。为了寻求高利率，Unix 经销商将价格抬得极高，PC 小用户就根本不能靠近它。曾经一度受到 Bell Labs 的许可而可以在大学中用于教学的 UNIX 源代码一直被小心地守卫着不需公开。对于广大的 PC 用户，软件行业的大型供应商始终没有给出有效的解决该问题的手段。

正在此时，出现了 MINIX 操作系统，并有一本详细的书本描述它的设计实现原理。由于 AST 的书写的非常详细，并且叙述有条有理，几乎全世界的计算机爱好者都在看这本书以理解操作系统的工作原理。其中也包括 Linux 系统的创始者 Linus Benedict Torvalds。

当时(1991 年)，Linus Benedict Torvalds 是赫尔辛基大学计算机科学系的二年级学生，也是一个自学 hacker。这个 21 岁的芬兰年轻人喜欢鼓捣计算机，测试计算机的能力和限制。但当时缺乏的是一个专业级的操作系统。MINIX 虽然很好，但只是一个用于教学目的简单操作系统，而不是一个强有力的实用操作系统。

到 1991 年，GNU 计划已经开发出了许多工具软件。最受期盼的 Gnu C 编译器已经出现，但还没有开发出免费的 GNU 操作系统。即使是 MINIX 也开始有了版权，需要购买才能得到源代码。而 GNU 的操作系统 HURD 一直在开发之中，但并不能在几年内完成。

对于 Linus 来说，已经不能等待了。从 1991 年 4 月份起，他开始酝酿并着手编制自己的操作系统。刚开始，他的目的很简单，只是为了学习 Intel 386 体系结构保护模式运行方式下的编程技术。但后来 Linux 的发展却完全改变了初衷。

1991 年初，Linux 开始在一台 386sx 兼容微机上学习 minix 操作系统。通过学习，他逐渐不能满足 minix 系统的现有性能，并开始酝酿开发一个新的免费操作系统。根据 Linus 在 comp.os.minix 新闻组上发布的消息，我们可以知道他逐步从学习 minix 系统到开发自己的 Linux 的过程。

Linus 第 1 次向 comp.os.minix 投递消息是在 1991 年 3 月 29 日。题目是"gcc on minix-386 doesn’t optimize",是有关 gcc 编译器在 minix-386 上运行的优化问题，由此可知，Linus 在 1991 年的初期已经开始深入研究了 minix 系统,并在这段时间有了改进 minix 操作系统的思想,而且在进一步学习 minix 系统中，逐步演变为想自己重新设计一个基于 Intel 80386 体系结构的新操作系统。

他在回答有人提出 minix 上的一个问题时，所说的第一句话是“阅读源代码”（“RTFSC (Read the F**ing Source Code :-)”）。他认为答案就在源程序中。这也说明了对于学习系统软件来说，你不光需要懂得系统的工作基本原理，还需要结合实际系统，学习实际系统的实现方法。因为理论毕竟是理论，其中省略了许多枝节，而这些枝节问题虽然没有太多的理论含量，但却是一个系统必要的组成部分，就向麻雀身上的一根羽毛。

从 1991 年的 4 月份开始，Linus 几乎花了全部时间研究 386-minix 系统(hack the kernel)，并且尝试着移植 GNU 的软件到该系统上(GNU gcc、bash、gdb 等)。并于 4 月 13 日在 comp.os.minix 上发布说自己已经成功地将 bash 移植到了 minix 上，而且已经爱不释手、不能离开这个 shell 软件了。

第一个与 Linux 有关的消息是在 1991 年 7 月 3 日在 comp.os.minix 上发布的(当然此时还不存在 Linux 这个名称，当时 Linus 的脑子里想的可能是 FREAX ☺，FREAX 的英文含义是怪诞的、怪物、异想天开等)。其中透露了他正在进行 Linux 系统的开发，并且在 Linux 最初的时候已经想到要实现与 POSIX（UNIX 的国际标准）的兼容问题了。

在 Linus 的下一发布的消息中(1991 年 8 月 25 日 comp.os.minix)，他向所有 minix 用户询问"What would you like to see in minix?"（“你最想在 minix 中见到什么？”），在该消息中他首次透露出正在开发一个(免费的)386(486)操作系统，并且说只是兴趣而已，代码不会很大，也不会象 GNU 的那样专

业。开发免费操作系统这个想法从 4 月份就开始酝酿了，希望大家反馈一些对于 minix 系统中喜欢那些特色不喜欢什么等信息，由于实际的和其它一些原因，新开发的系统刚开始与 minix 很象（并且使用了 minix 的文件系统）。并且已经成功地将 bash(1.08 版)和 gcc(1.40 版)移植到了新系统上，而且在过几个月就可以实用了。

最后，Linus 申明他开发的操作系统没有使用一行 minix 的源代码；而且由于使用了 386 的任务切换特性，所以该操作系统不好移植（没有可移植性），并且只能使用 AT 硬盘。对于 Linux 的移植性问题，Linus 当时并没有考虑。但是目前 Linux 几乎可以运行在任何一种硬件体系结构上。

到了 1991 年的 10 月 5 日，Linus 在 comp.os.minix 新闻组上发布消息，正式向外宣布 Linux 内核系统的诞生（Free minix-like kernel sources for 386-AT）。这段消息可以称为 Linux 的诞生宣言，并且一直广为流传。因此 10 月 5 日对 Linux 社区来说是一个特殊的日子，许多后来 Linux 的新版本发布时都选择了这个日子。所以 RedHat 公司选择这个日子发布它的新系统也不是偶然的。

1.1.6 Linux 操作系统版本的变迁

0.00 (1991.2-4?) 两个进程分别显示 AAA BBB
0.01 (1991.9?)第一个正式向外公布的 Linux 内核版本。
0.02 (1991.10.5)该版本以及 0.03 版是内部版本，目前已经无法找到。
0.03 (1991.10.5)
0.10 (1991.10)由 Ted Ts’o 发布的 Linux 内核版本。
0.11 (1991.12.8)基本可以正常运行的内核版本（也是本书着重注释的版本）。
0.12 (1992.1.15)主要加入对数学协处理器的软件模拟程序。
2.┌.　95(0.13) (1992.3.8) 开始加入虚拟文件系统思想的内核版本。
0.96 (1992.5.12)开始加入网络支持和虚拟文件系统 VFS。
0.97 (1992.8.1)
0.98 (1992.9.29)
0.99 (1992.12.13)
2.4.　(1994.3.14)
1.20 (1995.3.7)
2.0　(1996.2.9)
2.20 (1999.1.26)
2.40 (2001.1.4)
到现在为止，最新版是

内核版本号	发布日期		*代码总字节数*
2.4.20	2002.11.29		*26,200,000*
2.5.52	*2002.12.16*		*30,000,000*

将 Linux 系统 0.13 版内核直接改称 0.95 版，Linus 的意思是让大家不要觉得离 1.0 版还很遥远。同时，从 0.95 版开始，对内核的许多改进之处(补丁程序的提供)均以其他人为主了，而 Linus 的主要任务开始变成对内核的维护和决定是否采用某个补丁程序。

1.1.7 Linux 名称的来由

Linux 操作系统刚开始时并没有被称作 Linux，Linus 给他的操作系统取名为 FREAX，其英文含义是怪诞的、怪物、异想天开等意思。在他将新的操作系统上载到 ftp.funet.fi 服务器上时，管理员 Ari Lemke 很不喜欢这个名称。他认为既然是 Linus 的操作系统就取其谐音 Linux 作为该操作系统的目录吧，于是

Linux 这个名称就开始流传下来。

在 Linus 的自传《Just for Fun》一书中，Linus 解释说：

“坦白地说，我从来没有想到过要用 Linux 这个名称发布这个操作系统，因为这个名字有些太自负了。而我为最终发布版准备的是什么名字呢？Freax。实际上，内核代码中某些早期的 Makefile - 用于描述如何编译源代码的文件 - 文件中就已经包含有“Freax”这个名字了，大约存在了半年左右。但其实这也没什么关系，在当时还不需要一个名字，因为我还没有向任何人发布过内核代码。”

“而 Ari Lemke，他坚持要用自己的方式将内核代码放到 ftp 站点上，并且非常不喜欢 Freax 这个名字。他坚持要用现在这个名字(Linux)，我承认当时我并没有跟他多争论。但这都是他取的名字。所以我可以光明正大地说我并不自负，或者部分坦白地说我并没有本位主义思想。但我想好吧，这也是个好名字，而且以后为这事我总能说服别人，就象我现在做的这样。”

-- Linus Torvalds《Just for fun》第 84-88 页。

1.1.8 早期 Linux 系统开发的主要贡献者

从 Linux 的早期源代码中可知，Linux 系统的早期主要开发人员除了 Linus 本人以外，最著名的人员之一就是 Theodore Ts'o (Ted Ts'o)。他 1990 年毕业于 MIT 计算机科学专业。在大学时代他就积极参加学校中举办的各种学生活动。他喜欢烹饪、骑自行车，当然还有就是 hacking on Linux，后来他开始喜欢起业余无线电报运动。目前他在 IBM 工作从事系统编程及其它重要事务。他还是国际网络设计、操作、销售和研究者开放团体 IETF 成员。

Linux 在世界范围内的流行也有他很大的功劳。早在 Linux 操作系统刚问世时，他就怀着极大的热情为 linux 的发展提供了 maillist，几乎是在 Linux 刚开始发布起(1991 年开始)就一直为 Linux 做出贡献的人，也是最早向 Linux 内核添加程序的人(Linux 内核 0.10 版中的虚拟盘驱动程序 ramdisk.c 和内核内存分配程序 kmalloc.c)。直到目前仍然从事着与 Linux 有关的工作。他当时在北美洲地区最早设立了 linux 的 ftp 站点（tsx-11.mit.edu)，而且至今仍然为广大 linux 用户提供服务。他对 linux 作出的最大贡献之一是提出并实现了 ext2 文件系统。该文件系统已成为 linux 世界中事实上的文件系统标准。最近他又推出了 ext3 文件系统,大大提高了文件系统的稳定性和访问效率。作为对他的推崇,第 97 期(2002 年 5 月）的 linuxjournal 期刊将他作为了封面人物，并对他进行了采访。目前，他为 IBM linux 技术中心工作，并从事着有关 LSB(Linux Standard Base)等方面的工作。

Linux 社区中另一位著名人物是 Alan Cox。他原工作于英国威尔士斯旺西大学(Swansea University College)。刚开始他特别喜欢玩电脑游戏，尤其是 MUD（Multi-User Dungeon or Dimension，多用户网络游戏）。在 90 年代早期 games.mud 新闻组的 posts 中你可以找到他发表的大量 posts。他甚至为此还写了一偏 MUD 的发展史(rec.games.mud 新闻组，1992 年 3 月 9 日，A history of MUD)。由于 MUD 游戏与网络密切相关，慢慢的他对计算机网络开始感兴趣。为了玩游戏并提高电脑运行游戏的速度以及网络传输的速度，他开始接触各种类型的操作系统，为他的游戏选择一个最为满意的平台。由于没钱，即使 Minix 他都买不起，当 Linux 0.11 和 386BSD 发布时，他考虑良久总算买了一台 386SX 电脑。由于 386BSD 需要数学协处理器的支持，而 386SX 中是不带的，所以他安装了 Linux 系统。于是他开始学习带有免费源代码的 Linux。开始对 Linux 产生了兴趣，尤其是有关网络方面的实现。在关于 Linux 的单用户运行模式问题的讨论中，他甚至赞叹 Linux 实现的巧妙(beautifully)。

Linux 0.95 版发布之后，他开始为 Linux 系统编写补丁程序（修改程序)（记得他最早的两个补丁程序，都没有被 Linus 采纳)，成为 Linux 上 TCP/IP 网络代码的最早使用人之一。后逐渐加入 Linux 的开发队伍，并开始成为维护 Linux 内核源代码的主要负责人之一，也可以说成为 Linux 社团中在 Linus 之后最为重要的人物。以后 Microsoft 公司曾经邀请他加盟，但他却干脆地拒绝了。从 2001 年开始他负责维护 Linux 内核 2.4.x 的代码(Linus 主要负责开发最新开发版内核的研制(奇数版，比如 2.5.x 版)。

《内核骇客手册》一书的作者 Michael K. Johnson 也是最早接触 Linux 操作系统的人之一(从 0.97 版)。他还是著名 Linux 文档计划（Linux Document Project - LDP）的发起者之一。曾经在 Linux Journel

工作，现在 RedHat 公司工作。
Linux 系统并不是仅有这些中坚力量就能发展成今天这个样子的，还有许多计算机高手对 Linux 做出了极大的贡献，这里就不一一列举了。主要贡献者的具体名单可参见 Linux 内核中的 CREDITS 文件，其中以字母顺序列出了对 Linux 做出较大贡献的近 400 人的名单列表，包括他们的 email 地址和通信地址、主页以及主要贡献事迹等信息。

通过上述说明，我们可以对上述 Linux 的五大支柱归纳如下：

- UNIX 操作系统 -- UNIX 于 1969 年诞生在 Bell 实验室。Linux 就是 UNIX 的一种克隆系统。UNIX 的重要性就不用多说了。
- MINIX 操作系统 -- Minix 操作系统也是 UNIX 的一种克隆系统，它于 1987 年由著名计算机教授 Andrew S. Tanenbaum 开发完成。由于 MINIX 系统的出现并且提供源代码(只能免费用于大学内)在全世界的大学中刮起了学习 UNIX 系统旋风。Linux 刚开始就是参照 Minix 系统于 1991 年才开始开发。
- GNU 计划-- 开发 Linux 操作系统，以及 Linux 上所用大多数软件基本上都出自 GNU 计划。Linux 只是操作系统的一个内核，没有 GNU 软件环境(比如说 bash shell)，则 Linux 将寸步难行。
- POSIX 标准 -- 该标准在推动 Linux 操作系统以后朝着正规路上发展起着重要的作用。是 Linux 前进的灯塔。
- INTERNET -- 如果没有 Intenet 网，没有遍布全世界的无数计算机骇客的无私奉献，那么 Linux 最多只能发展到 0.13(0.95)版的水平。

1.2 内容综述

本文将主要对 Linux 的早期内核 0.11 版进行详细描述和注释。Linux-0.11 版本是在 1991 年 12 月 8 日发布的。在发布时包括以下文件：

```
bootimage.Z       - 具有美国键盘代码的压缩启动映像文件；
rootimage.Z       - 以 1200kB 压缩的根文件系统映像文件；
linux-0.11.tar.Z- 内核源代码文件；
as86.tar.Z        - linux bruce evans'二进制执行文件；
                    是 16 位的汇编程序和装入程序；
INSTALL-0.11      - 更新过的安装信息文件。
```

目前除了原来的 rootimage.Z 文件，其它四个文件均能找到。

本文主要详细分析 linux-0.11 内核中的所有源代码程序，对每个源程序文件都进行了详细注释，包括对 Makefile 文件的注释。分析过程主要是按照计算机启动过程进行的。因此分析的连贯性到初始化结束内核开始调用 shell 程序为止。其余的各个程序均针对其自身进行分析，没有连贯性，因此可以根据自己的需要进行阅读。但在分析时还是提供了一些应用实例。

所有的程序在分析过程中如果遇到作者认为是较难理解的语句时，将给出相关知识的详细介绍。比如，在阅读代码头一次遇到 C 语言内嵌汇编码时，将对 gnu C 语言的内嵌汇编语言进行较为详细的介绍；在遇到对中断控制器进行输入/输出操作时，将对 Intel 中断控制器（8259A）芯片给出详细的说明，并列出使用的命令和方法。这样做有助于加深对代码的理解，又能更好的了解所用硬件的使用方法，作者认为这种解读方法要比单独列出一章内容来总体介绍硬件或其它知识要效率高得多。

拿 Linux 0.11 版内核来“开刀”是为了提高我们认识 Linux 运行机理的效率。Linux-0.11 版整个内核源代码只有 325K 字节左右，其中包括的内容基本上都是 Linux 的精髓。而目前最新的 2.5.XX 版内核非常大，将近有 188 兆字节，即使你花一生的经历来阅读也未必能全部都看完。也许你要问“既然要从简入手，为什么不分析更小的 Linux 0.01 版内核源代码呢？它只有 240K 字节左右”主要原因是因为 0.01 版的内核代码有太多的不足之处，甚至还没有包括对软盘的驱动程序，也没有很好地涉及数学协处理器的使用以及对登陆程序的说明。并且其引导启动程序的结构也与目前的版本不太一样，而 0.11 版的引导启动程序结构则与现在的基本上是一样的。另外一个原因是可以找到 0.11 版早期的已经编译制作好的内核映像文件(bootimage)，可以用来进行引导演示。如果再配上简单的根文件系统映像文件(rootimage)，那么它就可以进行正常的运行了。

拿 Linux 0.11 版进行学习也有不足之处，比如该内核版本中尚不包括有关进程等待队列、TCP/IP 网络等方面的一些当前非常重要的代码，但好在 Linux 中的网络代码基本上是自成一体的，与内核机制关系不是非常大，因此可以在了解了 Linux 工作的基本原理之后再去分析这些代码。

本文对 Linux 内核中所有的代码都进行了说明。为了保持结构的完整性，对代码的说明是以内核中源代码的组成结构来进行的，基本上是以每个源代码中的目录为一章内容进行介绍。介绍的源程序文件的次序可参见前面的文件列表索引。整个 Linux 内核源代码的目录结构如下列表 1.1 所示。所有目录结构均是以 linux 为当前目录。

列表 1.1 Linux/目录

Name	Size	Last modified (GMT)	Description
boot/		1991-12-05 22:48:49	
fs/		1991-12-08 14:08:27	
include/		1991-09-22 19:58:04	
init/		1991-12-05 19:59:04	
kernel/		1991-12-08 14:08:00	
lib/		1991-12-08 14:10:00	
mm/		1991-12-08 14:08:21	
tools/		1991-12-04 13:11:56	
Makefile	2887 bytes	1991-12-06 03:12:46	

本章主要说明内核程序包中 Linux/目录下的所看到的第一个文件，也即内核代码的总体 Makefile 文件的内容。

第 2 章将详细注释 boot/目录下的三个汇编程序，其中包括磁盘引导程序 bootsect.s、32 位运行启动代码程序 head.s 和获取 BIOS 中参数的 setup.s 汇编程序。

第 3 章主要介绍 init/目录中内核系统的初始化程序 main.c。它是内核完成所有初始化工作并进入正常运行的关键地方。在介绍该程序时将需要查看其所调用的其它程序，因此对后续章节的阅读可以按照这里调用的顺序进行。当你能真正看懂直到 main.c 程序为止的所有程序时，你应该已经对 Linux 内核有了一定的了解，可以说已经有一半入门了☺，但你还需要对文件系统、系统调用、各种驱动程序等进行更深一步的阅读。

第 4 章主要介绍 kenel/目录中的所有程序。此时你应该已经对其中的一些重要程序有所了解。

第 5 章介绍内核源代码 fs/目录中的文件系统程序，在看这章内容时建议你能够暂停一下而去阅读 Andrew S. Tanenbaum 的《操作系统设计与实现》一书中有关 minix 文件系统的章节，因为最初的 Linux 系统是只支持 minix 一种文件系统，Linux 0.11 版也不例外。

第 6 章解说 mm/目录中的内存管理程序。要透彻地理解这方面的内容，需要对 Intel 80X86 微处理器的保护模式运行方式有足够的理解，因此本章在适当的地方包含有较为完整的有关 80X86 保护模式运行方式的说明，这些知识基本上是从 Intel 80386 程序员编程手册(Intel 80386 Programmer's Reference Manual) 中摘录下来的，但在此是以源代码中的运用实例为对象进行解说的，因此应该能更好地理解它的原理。

第 7 章是对 include/目录中的所有头文件进行详细说明，基本上对每一个定义、每一个常量或数据结构都进行了详细注释。为了便于在阅读时参考查阅，本书在附录中还对一些经常要用到的重要的数据结构和变量进行了归纳注释，但这些内容其实都能在这一章中找到。

第 8 章介绍 Linux 0.11 版内核源代码中 lib/目录中的所有文件。由于这个版本较低，所以这里的内容并不是很多，可以很快地看完。这也是我们为什么选择 0.11 版的原因之一。

第 9 章介绍 tools/目录下的 build.c 程序。这个程序并不会包括在编译生成的内核映像(image)文件中，它仅用于将内核中的磁盘引导程序块与其它主要内核模块连接成一个完整了内核映像（kernel image）文件。

为了便于查阅，在本书的附录中还单独列出了内核中要用到的有关 PC 机硬件方面的信息。在参考文献中，我们仅给出了在阅读源代码时可以参考的书籍、文章等信息，并没有包罗万象地给出一大堆的繁杂凌乱的文献列表。比如在引用 Linux 文档项目 LDP（Linux Document Project）中的文件时，我们会明确地列出具体需要参考哪一篇 HOWTO 文章，而并不是仅仅给出 LDP 的网站地址了事。

在对每个程序进行解说时，我们首先简单说明程序的主要用途和目的、输入输出参数以及与其它程序的关系，然后列出程序的完整代码并在其中对代码进行详细注释，注释时对原程序代码或文字不作任何方面的改动或删除，因为 C 语言是一种英语类语言，程序中原有的少量英文注释对常数符号、变量名等也提供了不少有用的信息。在代码之后是对程序更为深入的解剖，并对代码中出现的一些语言或硬件方面的相关知识进行说明。如果在看完这些信息后回头再浏览一遍程序，你会有更深一层的体会。

最后要说明的是当你已经完全理解了本文说解说的一切时，并不代表你已经成为一个 Linux 行家了，你只是刚刚踏上 Linux 的征途，具有了一定的成为一个 Linux GURU 的初步知识。这时你应该去阅读更多的源代码，最好是循序渐进地从 1.0 版本开始直到最新的正在开发中的奇数编号的版本。在撰写这本书时最新的 Linux 内核是 2.5.44 版。当你能快速理解这些开发中的最新版本甚至能提出自己的建议和补丁（patch）程序时，我也甘拜下风了☺。

最后祝你征途愉快！

第 2 章 linux 内核体系结构

本章首先概要介绍了早期 Linux 内核的编制模式和体系结构，然后详细描述了 linux 内核源代码目录中组织形式以及子目录中各个代码文件的主要功能以及基本调用的层次关系。接下来就直接切入正题，从内核源文件 linux/目录下的第一个文件 Makefile 开始，对每一行代码进行详细注释说明。

一个完整可用的操作系统主要由 4 部分组成：硬件、操作系统内核、操作系统服务和用户应用程序，见图 2.1 所示。用户应用程序是指那些字处理程序、Internet 浏览器程序或用户自行编制的各种应用程序；操作系统服务程序是指那些向用户所提供的服务被看作是操作系统的部分功能的程序。在 Linux 操作系统上，这些程序包括 X 窗口系统、shell 命令解释系统以及那些内核编程接口等系统程序；操作系统内核程序即是本书所感兴趣的部分，它主要用于对硬件资源的抽象和访问调度。

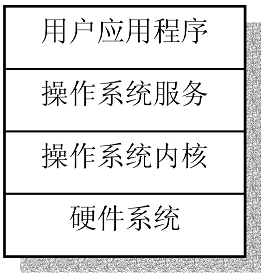


图 2.1 操作系统组成部分。

Linux 内核的主要用途就是为了与计算机硬件进行交互，实现对硬件部件的编程控制和接口操作，调度对硬件资源的访问，并为计算机上的用户程序提供一个高级的执行环境和对硬件的虚拟接口。
在本章内容中，我们首先基于 Linux 0.11 版的内核源代码，简明地描述 Linux 内核的基本体系结构、主要构成模块。然后对源代码中出现的几个重要数据结构进行说明。最后描述了构建 Linux 0.11 内核编译实验环境的方法。

2.1 Linux 内核模式

目前，操作系统内核的结构模式主要可分为整体式的单内核模式和层次式的微内核模式。而本书所注释的 Linux 0.11 内核，则是采用了单内核模式。单内核模式的主要优点是内核代码结构紧凑、执行速度快，不足之处主要是层次结构性不强。

在单内核模式的系统中，操作系统所提供服务的流程为：应用主程序使用指定的参数值执行系统调用指令(int x80)，使 CPU 从用户态（User Mode）切换到核心态（Kernel Model），然后操作系统根据具体的参数值调用特定的系统调用服务程序，而这些服务程序则根据需要再底层的一些支持函数以完成特定的功能。在完成了应用程序所要求的服务后，操作系统又从核心态切换回用户态，返回到应用程序中继续执行后面的指令。因此概要地讲，单内核模式的内核也可粗略地分为三个层次：调用服务的主程序层、执行系统调用的服务层和支持系统调用的底层函数。见图 2.2 所示。

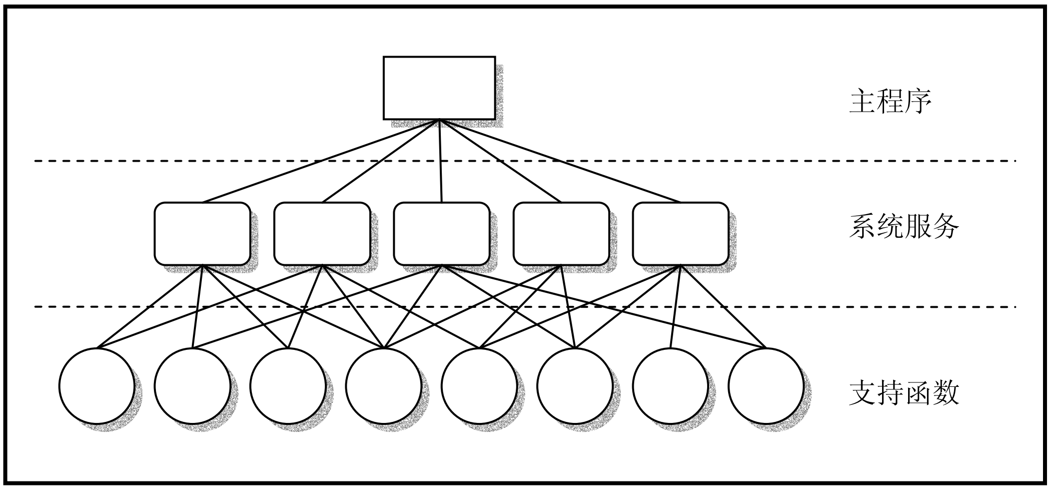


图 2.2 单内核模式的简单结构模型

2.2 Linux 内核系统体系结构

Linux 内核主要由 5 个模块构成，它们分别是：进程调度模块、内存管理模块、文件系统模块、进程间通信模块和网络接口模块。

进程调度模块用来负责控制进程对 CPU 资源的使用。所采取的调度策略是各进程能够公平合理地访问 CPU，同时保证内核能正时地执行硬件操作。内存管理模块用于确保所有进程能够安全地共享机器主内存区，同时，内存管理模块还支持虚拟内存管理方式，使得 Linux 支持进程使用比实际内存空间更多大的内存容量。并可以利用文件系统把暂时不用的内存数据块会被交换到外部存储设备上去，当需要时再交换回来。文件系统模块用于支持对外部设备的驱动和存储。虚拟文件系统模块通过向所有的外部存储设备提供一个通用的文件接口，隐藏了各种硬件设备的不同细节。从而提供并支持与其它操作系统兼容的多种文件系统格式。进程间通信模块子系统用于支持多种进程间的信息交换方式。网络接口模块提供对多种网络通信标准的访问并支持许多网络硬件。

这几个模块之间的依赖关系见图 2.3 所示。其中的连线代表它们之间的依赖关系，虚线和虚框部分表示 Linux 0.11 中还未实现的部分（从 Linux 0.95 版才开始逐步实现虚拟文件系统，而网络接口的支持到 0.96 版才有）。

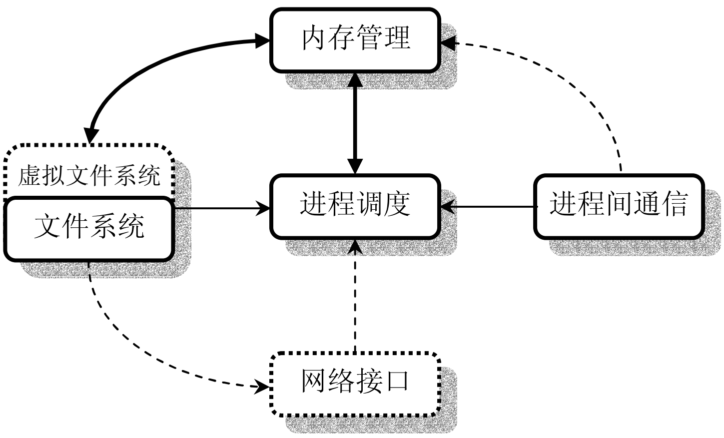


图 2.3 Linux 内核系统模块结构及相互依赖关系

由图可以看出，所有的模块都与进程调度模块存在依赖关系。因为它们都需要依靠进程调度程序来

挂起（暂停）或重新运行它们的进程。通常，一个模块会在等待硬件操作期间被挂起，而在操作完成后才可继续运行。例如，当一个进程试图将一数据块写到软盘上去时，软盘驱动程序就可能在启动软盘旋转期间将该进程置为挂起等待状态，而在软盘进入到正常转速后再使得该进程能继续运行。另外 3 个模块也是由于类似的原因而与进程调度模块存在依赖关系。

其它几个依赖关系有些不太明显，但同样也很重要。进程调度子系统需要使用内存管理器来调整一特定进程所使用的物理内存空间。进程间通信子系统则需要依靠内存管理器来支持共享内存通信机制。这种通信机制允许两个进程访问内存的同一个区域以进行进程间信息的交换。虚拟文件系统也会使用网络接口来支持网络文件系统（NFS），同样也能使用内存管理子系统来提供内存虚拟盘（ramdisk）设备。而内存管理子系统也会使用文件系统来支持内存数据块的交换操作。

除了这些图中已经给出的依赖关系以外，所有这些模块还会依赖于内核中的通用资源。这些资源包括内核所有子系统都会调用的内存分配和收回函数、打印警告或出错信息函数以及一些系统调试函数。

2.3 Linux 内核源代码的目录结构

由于 Linux 内核是一种单内核模式的系统，因此，内核中所有的程序几乎都有紧密的联系，它们之间的依赖和调用关系非常密切。所以在阅读一个源代码文件时往往需要参阅其它相关的文件。因此有必要在开始阅读内核源代码之前，先熟悉一下源代码文件的目录结构和安排。

这里我们首先列出 Linux 内核完整的源代码目录，包括其中的子目录。然后逐一介绍各个目录中所包含程序的主要功能，使得整个内核源代码的安排形式能在我们的头脑中建立起一个大概的框架，以便于下一章开始的源代码阅读工作。

当我们使用 tar 命令将 linux-0.11.tar.gz 解开时，内核源代码文件被放到了 linux 目录中。其中的目录结构为：

```
linux
├── boot                系统引导汇编程序
├── fs                  文件系统
├── include             头文件(*.h)
│    ├── asm            与 CPU 体系结构相关的部分
│    ├── linux          Linux 内核专用部分
│    └── sys            系统数据结构部分
├── init                内核初始化程序
├── kernel              内核进程调度、信号处理、系统调用等程序
│    ├── blk_drv        块设备驱动程序
│    ├── chr_drv        字符设备驱动程序
│    └── math           数学协处理器仿真处理程序
├── lib                 内核库函数
├── mm                  内存管理程序
└── tools               生成内核 Image 文件的工具程序
```

该内核版本的源代码目录中含有 14 个子目录，总共包括 102 个代码文件。下面逐个对这些子目录中的内容进行描述。

2.3.1 内核主目录 linux

linux 目录是源代码的主目录，在该主目录中除了包括所有的 14 个子目录以外，还含有唯一的一个 makefile 文件。该文件是编译辅助工具软件 make 的参数配置文件。make 工具软件的主要用途是通过识别哪些文件已被修改过，从而自动地决定在一个含有多个源程序文件的程序系统中哪些文件需要被重新编译。因此，make 工具软件是程序项目的管理软件。

linux 目录下的这个 makefile 文件还嵌套地调用了所有子目录中包含的 makefile 文件，这样，当 linux 目录（包括子目录）下的任何文件被修改过时，make 都会对其进行重新编译。因此为了编译整个内核所有的源代码文件，只要在 linux 目录下运行一次 make 软件即可。

2.3.2 引导启动程序目录 boot

boot 目录中含有 3 个汇编语言文件，是内核源代码文件中最先被编译的程序。这 3 个程序完成的主要功能是当计算机加电时引导内核启动，将内核代码加载到内存中，并做一些进入 32 位保护运行方式前的系统初始化工作。其中 bootsect.s 和 setup.s 程序需要使用 as86 软件来编译，使用的是 as86 的汇编语言格式（与微软的类似），而 head.s 需要用 GNU as 来编译，使用的是 AT&T 格式的汇编语言。这两种汇编语言在下一章的代码注释里以及代码列表后面的说明中会有简单的介绍。

bootsect.s 程序是磁盘引导块程序，编译后会驻留在磁盘的第一个扇区中（引导扇区，0 磁道（柱面），0 磁头，第 1 个扇区）。在 PC 机加电 ROM BIOS 自检后，将被 BIOS 加载到内存 0x7C00 处进行执行。

setup.s 程序主要用于读取机器的硬件配置参数，并把内核模块 system 移动到适当的内存位置处。

head.s 程序会被编译连接在 system 模块的最前部分，主要进行硬件设备的探测设置和内存管理页面的初始设置工作。

2.3.3 文件系统目录 fs

是文件系统实现程序的目录，共包含 17 个 C 语言程序。这些程序之间的主要引用关系见图 2.4 所示图中每个方框代表一个文件，从上到下按基本按引用关系放置。其中各文件名均略去了后缀.c，虚框中是的程序文件不属于文件系统，带箭头的线条表示引用关系，粗线条表示有相互引用关系。

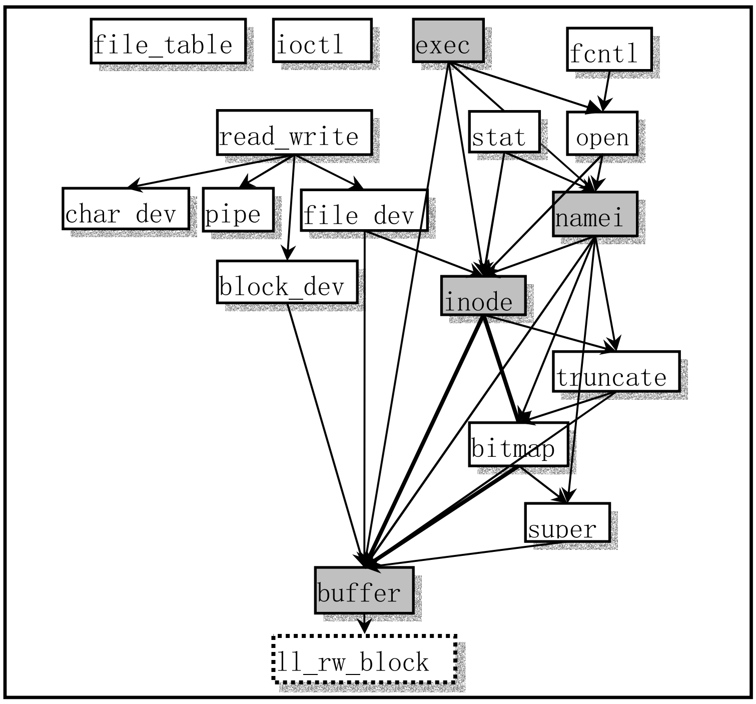


图 2.4 fs 目录中各程序中函数之间的引用关系。

在 file_table.c 文件中，目前仅定义了一个文件句柄（描述符）结构数组。ioctl.c 文件将引用 kernel/chr_dev/tty.c 中的函数，实现字符设备的 io 控制功能。exec.c 程序主要包含一个执行程序函数 do_execve()，它是所有 exec()函数簇中的主要函数。fcntl.c 程序用于实现文件 i/o 控制的系统调用函数。read_write.c 程序用于实现文件读/写和定位三个系统调用函数。stat.c 程序中实现了两个获取文件状态的系统调用函数。open.c 程序主要包含实现修改文件属性和创建与关闭文件的系统调用函数。

char_dev.c 主要包含字符设备读写函数 rw_char()。pipe.c 程序中包含管道读写函数和创建管道的系统调用。file_dev.c 程序中包含基于 i 节点和描述符结构的文件读写函数。namei.c 程序主要包括文件系统中目录名和文件名的操作函数和系统调用函数。block_dev.c 程序包含块数据读和写函数。inode.c 程序中包含针对文件系统 i 节点操作的函数。truncate.c 程序用于在删除文件时释放文件所占用的设备数据空间。bitmap.c 程序用于处理文件系统中 i 节点和逻辑数据块的位图。super.c 程序中包含对文件系统超级块的处理函数。buffer.c 程序主要用于对内存高速缓冲区进行处理。虚框中的 ll_rw_block 是块设备的底层读函数，它并不在 fs 目录中，而是 kernel/blk_dev/ll_rw_block.c 中的块设备读写驱动函数。放在这里只是让我们清楚的看到，文件系统对于块设备中数据的读写，都需要通过高速缓冲区与块设备的驱动程序（ll_rw_block()）来操作来进行，文件系统程序集本身并不直接与块设备的驱动程序打交道。

在对程序进行注释过程中，我们将另外给出这些文件中各个主要函数之间的调用层次关系。

2.3.4 头文件主目录 include

头文件目录中总共有 32 个.h 头文件。其中主目录下有 13 个，asm 子目录中有 4 个，linux 子目录中有 10 个，sys 子目录中有 5 个。这些头文件各自的功能见如下简述，具体的作用和所包含的信息请参见对头文件的注释一章。

<a.out.h> a.out 头文件，定义了 a.out 执行文件格式和一些宏。
<const.h> 常数符号头文件，目前仅定义了 i 节点中 i_mode 字段的各标志位。
<ctype.h> 字符类型头文件。定义了一些有关字符类型判断和转换的宏。
<errno.h> 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
<fcntl.h> 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
<signal.h> 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。
<stdarg.h> 标准参数头文件。以宏的形式定义变量参数列表。主要说明了-个类型（va_list）和三个宏（va_start, va_arg 和 va_end)，用于 vsprintf、vprintf、vfprintf 函数。
<stddef.h> 标准定义头文件。定义了 NULL, offsetof(TYPE, MEMBER)。
<string.h> 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
<termios.h> 终端输入输出函数头文件。主要定义控制异步通信口的终端接口。
<time.h> 时间类型头文件。其中最主要定义了 tm 结构和一些有关时间的函数原形。
<unistd.h> Linux 标准头文件。定义了各种符号常数和类型，并申明了各种函数。如定义了 __LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()等。
<utime.h> 用户时间头文件。定义了访问和修改时间结构以及 utime()原型。

2.3.4.1 体系结构相关头文件子目录 include/asm

这些头文件主要定义了一些与 CPU 体系结构密切相关的数据结构、宏函数和变量。共 4 个文件。

<asm/io.h> io 头文件。以宏的嵌入汇编程序形式定义对 io 端口操作的函数。
<asm/memory.h> 内存拷贝头文件。含有 memcpy()嵌入式汇编宏函数。
<asm/segment.h> 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

<asm/system.h> 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

2.3.4.2 Linux 内核专用头文件子目录 include/linux

<linux/config.h> 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）可选项。
<linux/fdreg.h> 软驱头文件。含有软盘控制器参数的一些定义。
<linux/fs.h> 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
<linux/hdreg.h> 硬盘参数头文件。定义访问硬盘寄存器端口，状态码，分区表等信息。
<linux/head.h> head 头文件，定义了段描述符的简单结构，和几个选择符常量。
<linux/kernel.h> 内核头文件。含有一些内核常用函数的原形定义。
<linux/mm.h> 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
<linux/sched.h> 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
<linux/sys.h> 系统调用头文件。含有 72 个系统调用 C 函数处理程序,以'sys_'开头。
<linux/tty.h> tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。

2.3.4.3 系统专用数据结构子目录 include/sys

<sys/stat.h> 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。
<sys/times.h> 定义了进程中运行时间结构 tms 以及 times()函数原型。
<sys/types.h> 类型头文件。定义了基本的系统数据类型。
<sys/utsname.h> 系统名称结构头文件。
<sys/wait.h> 等待调用头文件。定义系统调用 wait()核 waitpid()及相关常数符号。

2.3.5 内核初始化程序目录 init

该目录中仅包含一个文件 main.c。用于执行内核所有的初始化工作，然后移到用户模式创建新进程，并在控制台设备上运行 shell 程序。

程序首先根据机器内存的多少对缓冲区内存容量进行分配，如果还设置了要使用虚拟盘，则在缓冲区内存后面也为它留下空间。之后就进行所有硬件的初始化工作，包括人工创建第一个任务（task 0），并设置了中断允许标志。在执行从核心态移到用户态之后，系统第一次调用创建进程函数 fork()，创建出一个用于运行 init()的进程，在该子进程中，系统将进行控制台环境设置，并且在生成一个子进程用来运行 shell 程序。

2.3.6 内核程序主目录 kernel

linux/kernel 目录中共包含 12 个代码文件和一个 Makefile 文件，另外还有 3 个子目录。由于这些文件中代码之间调用关系复杂，因此这里就不详细列出各文件之间的引用关系图，但仍然可以进行大概分类，见图 2.5 所示。

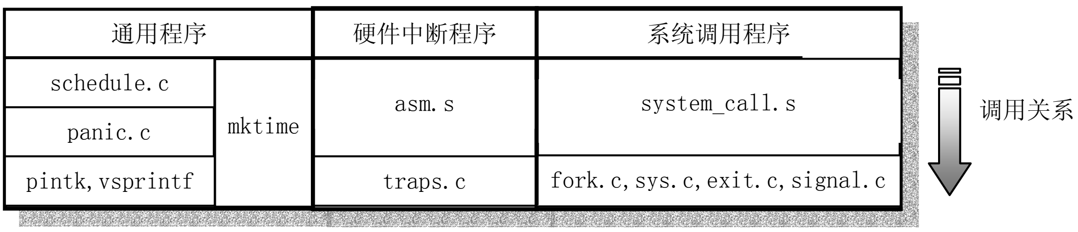


图 2.5 各文件的调用层次关系

asm.s 程序是用于处理系统硬件异常所引起的中断，对各硬件异常的实际处理程序则是在 traps.c

文件中，在各个中断处理过程中，将分别调用 traps.c 中相应的 C 语言处理函数。

exit.c 程序主要包括用于处理进程终止的系统调用。包含进程释放、会话（进程组）终止和程序退出处理函数以及杀死进程、终止进程、挂起进程等系统调用函数。

fork.c 程序给出了 sys_fork()系统调用中使用了两个 C 语言函数：find_empty_process()和 copy_process()。

mktime.c 程序包含一个内核使用的时间函数 mktime()，用于计算从 1970 年 1 月 1 日 0 时起到开机当日的秒数，作为开机秒时间。仅在 init/main.c 中被调用一次。

panic.程序包含一个显示内核出错信息并停机的函数 panic()。

printk.c 程序包含一个内核专用信息显示函数 printk()。

sched.c 程序中包括有关调度的基本函数(sleep_on、wakeup、schedule 等)以及一些简单的系统调用函数。另外还有几个与定时相关的软盘操作函数。

signal.c 程序中包括了有关信号处理的 4 个系统调用以及一个在对应的中断处理程序中处理信号的函数 do_signal()。

sys.c 程序包括很多系统调用函数，其中有些还没有实现。

system_call.s 程序实现了 linux 系统调用（int 0x80）的接口处理过程，实际的处理过程则包含在各系统调用相应的 C 语言处理函数中，这些处理函数分布在整个 linux 内核代码中。

vsprintf.c 程序实现了现在已经归入标准库函数中的字符串格式化函数。

2.3.6.1 块设备驱动程序子目录 kernel/blk_dev

该子目录共包含 4 个 c 文件和 1 个头文件。头文件 blk.h 由于是块设备程序专用的，所以与 C 文件放在一起。这几个文件之间的大致关系，见图 2.6 所示。

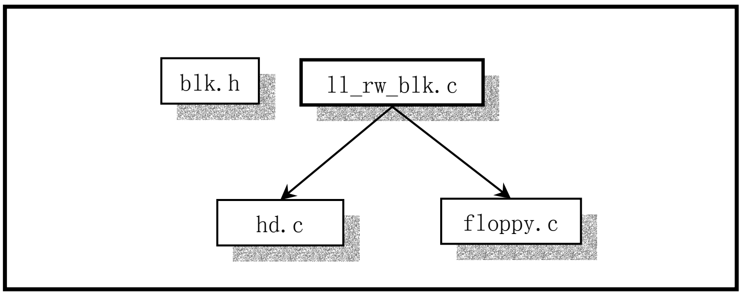


图 2.6 blk_dev 目录中文件的层次关系。

blk.h 中定义了 3 个 C 程序中共用的块设备结构和数据块请求结构。hd.c 程序主要实现对硬盘数据块进行读/写的底层驱动函数，主要是 do_hd__request()函数；floppy.c 程序中主要实现了对软盘数据块的读/写驱动函数，主要是 do_fd_request()函数。ll_rw_blk.c 中程序实现了低层块设备数据读/写函数 ll_rw_block()，内核中所有其它程序都是通过该函数对块设备进行数据读写操作。你将看到该函数在许多访问块设备数据的地方被调用，尤其是在高速缓冲区处理文件 fs/buffer.c 中。

2.3.6.2 字符设备驱动程序子目录 kernel/chr_dev

字符设备程序子目录共含有 4 个 C 语言程序和 2 个汇编程序文件。这些文件实现了对串行端口 rs-232、tty、键盘和终端控制台设备的驱动。下图（图 2.7）是这些文件之间的大致调用层次关系。

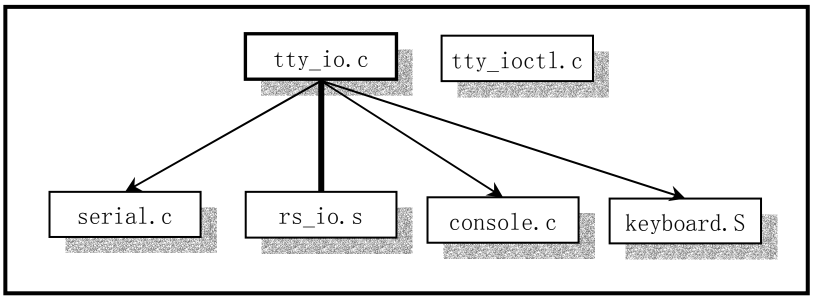


图 2.7 字符设备程序之间的关系示意图。

tty_io.c 程序中包含 tty 字符设备读函数 tty_read()和写函数 tty_write()，另外还包括在串行中断处理过程中调用的 C 函数 do_tty_interrupt()，该函数将会在中断类型为读字符的处理中被调用。console.c 文件主要包含控制台初始化程序和控制台写函数 con_write()，用于被 tty 设备调用。还包含对显示器和键盘中断的初始化设置程序 con_init()。

rs_io.s 汇编程序用于实现两个串行接口的中断处理程序。该中断处理程序会根据从中断标识寄存器（端口 0x3fa 或 0x2fa）中取得的 4 种中断类型分别进行处理，并在处理中断类型为读字符的代码中调用 do_tty_interrupt()。

serial.c 用于对异步串行通信芯片 UART 进行初始化操作，并设置两个通信端口的中断向量。另外还包括 tty 用于往串口输出的 rs_write()函数。

tty_ioctl.c 程序实现了 tty 的 io 控制接口函数 tty_ioctl()以及对 termio(s)终端 io 结构的读写函数，并会在实现系统调用 sys_ioctl()的 fs/ioctl.c 程序中被调用。

keyboard.S 程序主要实现了键盘中断处理过程 keyboard_interrupt。

2.3.6.3 协处理器仿真和操作程序子目录 kernel/math

该子目录中目前仅有一个 C 程序 math_emulate.c。其中的 math_emulate()函数是中断 int7 的中断处理程序调用的 C 函数。当机器中没有数学协处理器，而 CPU 却又执行了协处理器的指令时，就会引发该中断。因此，使用该中断就可以用软件来仿真协处理器的功能。本书所讨论的内核版本还没有包含有关协处理器的仿真代码。本程序中只是打印一条出错信息，并向用户程序发送一个协处理器错误信号 SIGFPE。

2.3.7 内核库函数目录 lib

内核库函数主要用于用户编程调用，是编译系统标准库的接口函数之一。其中共有 12 个 C 语言文件，除了一个由 tytso 编制的 malloc.c 程序较长以外，其它的程序很短，有的只有一二行代码。
这些文件中主要包括有退出函数_exit()、关闭文件函数 close(fd)、复制文件描述符函数 dup()、文件打开函数 open()、写文件函数 write()、执行程序函数 execve()、内存分配函数 malloc()、等待子进程状态函数 wait()、创建会话系统调用 setsid()以及在 include/string.h 中实现的所有字符串操作函数。

2.3.8 内存管理程序目录 mm

该目录包括 2 个代码文件。

page.s 文件包括内存页面异常中断（int 14）处理程序，主要用于处理程序由于缺页而引起的页异常中断和访问非法地址而引起的页保护。

memory.c 程序包括对内存进行初始化的函数 mem_init()，由 page.s 的内存处理中断过程调用的 do_no_page()和 do_wp_page()函数。在创建新进程而执行复制进程操作时，即使用该文件中的内存处理函数来分配管理内存空间。

2.3.9 编译内核工具程序目录 tools

该目录下的 build.c 程序用于将 Linux 各个目录中被分别编译生成的目标代码连接合并成一个可运行的内核映象文件 image。其具体的功能可参见下一章内容。

2.4 Linux 内核的编译实验环境

最初的 Linux 操作系统内核是在 Minix 1.5.10 操作系统的扩展版本 Minix-i386 上交叉编译开发的。Minix 1.5.10 该版本的操作系统是随 A.S. Tanenbaum 的《Minix 设计与实现》一书第 1 版一起由 Prentice Hall 发售的。该版本的 Minix 虽然可以运行在 80386 及其兼容微机上，但并没有利用 80386 的 32 位机制。为了能在该系统上进行 32 位操作系统的开发，Linus 使用了 Bruce Evans 的补丁程序将其升级为 MINIX-386，并把 GNU 的系列开发工具 gcc、gld、emacs、bash 等移植到 Minix-386 上。在这个平台上，Linus 进行交叉编译，开发出 Linux 0.01、0.11、0.12 等版本的内核。作者曾根据 Linux 的邮件列表的文章介绍，建立起了当时的开发平台，顺利地编译出 Linux 的早期版本内核（见 http://oldlinux.org 论坛中的介绍）。

但由于 Minix 1.5.10 早已过时，而且该开发平台的建立非常烦琐，因此这里只简单介绍一下如何修改 Linux 0.11 版内核源代码，使其能在目前常用的 RedHat 9 操作系统标准的编译环境下进行编译，并生成可运行的启动映像文件 boot-iamge。这里仅给出主要的修改方面，所有的修改之处可使用工具 diff 来比较修改后和未修改前的代码，找出其中的区别。假如，未修改过的代码在 linux 目录中，修改过的代码在 linux-mdf 中，则需要执行下面的命令：

```
diff -r linux linux-mdf > dif.out
```

其中文件 dif.out 中即包含代码中所有修改过的地方。已经修改好并能在 RedHat 9 下编译的 Linux 0.11 内核源代码可以从下面地址处下载：

http://oldlinux.org/Linux.old/kernel/linux-0.11-030920.tar.gz

2.4.1 修改 makefile 文件

在 Linux 0.11 内核代码文件中，几乎每个子目录中都包括一个 makefile 文件，需要对它们都进行以下修改：

a. 将 gas =>as, gld=>ld。现在 gas 和 gld 已经直接改名称为 as 和 ld 了。

b. as(原 gas)已经不用-c 选项，所以要将 Makefile, 因此需要去掉其-c 编译选项。在内核主目录 Linux 下 makefile 文件中，是在 34 行上。

c. 去掉 gcc 的编译标志选项：-fcombine-regs、-mstring-insns 以及所有子目录中 Makefile 中的这两个选项。在 94 年的 gcc 手册中就已找不到-fcombine-regs 选项，而-mstring-insns 是 Linus 自已对 gcc 的修改增加的选项，所以你我的 gcc 中肯定不包括这个优化选项。

d. 在 gcc 的编译标志选项中，增加-m386 选项。这样在 RedHat 9 下编译出的内核映像文件中就不含有 80486 及以上 CPU 的指令，因此该内核就可以运行在 80386 机器上。

2.4.2 修改汇编程序中的注释

as86 编译程序不能识别 c 语言的注释语句，因此需要使用!注释掉 boot/bootsect.s 文件中的 C 注释语句。

2.4.3 内存位置对起 align 值的修改

在 boot 目录下的三个汇编程序中，align 语句使用的方法目前已经改变。原来 align 后面带的数值是指对起内存位置的幂次值，而现在则需要直接给出对起的整数地址值。因此，原来的语句：

```
.align 3
```

需要修改成(2 的 3 次幂值 2^3=8)：

```
.align 8
```

2.4.4 修改嵌入宏汇编程序

由于对 as 的不断改进，目前其自动化程度越来越高，因此已经不需要人工指定一个变量需使用的 CPU 寄存器。因此内核代码中的__asm__("ax")需要全部去掉。例如 fs/bitmap.c 文件的第 20 行、26 行上，fs/namei.c 文件的第 65 行上等。

在嵌入汇编代码中，另外还需要去掉所有对寄存器内容无效的声明。例如 include/string.h 中第 84 行：

```
:"si","di","ax","cx");
```

需要修改成：

```
: );
```

2.4.5 c 程序变量在汇编语句中的引用表示

在开发 Linux 0.11 时所用的汇编器，在引用 C 程序中的变量时需要在变量名前加一下划线字符'_'，而目前的 gcc 编译器可以直接识别使用这些汇编中引用的 c 变量，因此需要将汇编程序（包括嵌入汇编语句）中所有 c 变量之前的下划线去掉。例如 boot/head.s 程序中第 15 行语句：

```
.globl _idt,_gdt,_pg_dir,_tmp_floppy_area
```

需要改成：

```
.globl idt,gdt,pg_dir,tmp_floppy_area
```

第 31 行语句：

```
lss _stack_start,%esp
```

需要改成：

```
lss stack_start,%esp
```

2.5 Makefile 文件

从本节起，我们开始对内核源代码文件进行注释。首先注释 linux 目录下遇到的第一个文件 Makefile。后续章节将按照这里类似的描述结构进行注释。

Makefile 文件相当于程序编译过程中的批处理文件。是工具程序 make 运行时的输入数据文件。只要在含有 Makefile 的当前目录中键入 make 命令，它就会依据 Makefile 文件中的设置对源程序或目标代码文件进行编译、链节或进行安装等活动。

make 工具程序能自动地确定一个大程序系统中那些程序文件需要被重新编译，并发出命令对这些程序文件进行编译。在使用 make 之前，需要编写 Makefile 信息文件，该文件描述了整个程序包中各程序之间的关系，并针对每个需要更新的文件给出具体的控制命令。通常，执行程序是根据其目标文件进行更新的，而这些目标文件则是由编译程序创建的。一旦编写好一个合适的 Makefile 文件，那么在你每次修改过程序系统中的某些源代码文件后，执行 make 命令就能进行所有必要的重新编译工作。make 程序是使用Makefile数据文件和代码文件的最后修改时间(last-modification time)来确定那些文件需要进

行更新，对于每一个需要更新的文件它会根据 Makefile 中的信息发出相应的命令。在 Makefile 文件中，开头为'#'的行是注释行。文件开头部分的'='赋值语句定义了一些参数或命令的缩写。

这个 Makefile 文件的主要作用是指示 make 程序最终使用独立编译连接成的 tools/目录中的 build 执行程序将所有内核编译代码连接和合并成一个可运行的内核映像文件 image。具体是对 boot/中的 bootsect.s、setup.s 使用 8086 汇编器进行编译，分别生成各自的执行模块。再对源代码中的其它所有程序使用 GNU 的编译器 gcc/gas 进行编译，并连接成模块 system。再用 build 工具将这三块组合成一个内核映象文件 image. 基本编译连接/组合结构如下图 2.8 所示。

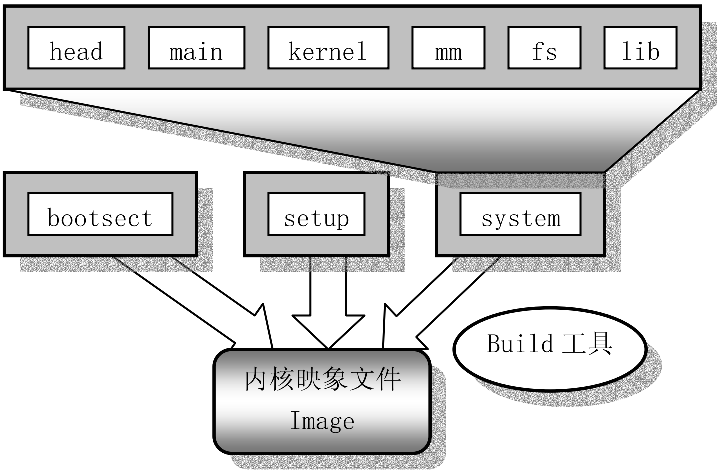


图 2.8 内核编译连接/组合结构

2.6 linux/Makefile 文件

列表 2.1 linux/Makefile 文件

```
#
# if you want the ram-disk device, define this to be the # 如果你要使用 RAM 盘设备的话，就
# size in blocks.                                        # 定义块的大小。
#
RAMDISK = #-DRAMDISK=512

AS86    =as86 -0 -a       # 8086 汇编编译器和连接器，见列表后的介绍。后带的参数含义分别
LD86    =ld86 -0          # 是：-0 生成 8086 目标程序；-a 生成与 gas 和 gld 部分兼容的代码。

AS      =gas              # GNU 汇编编译器和连接器，见列表后的介绍。
LD      =gld
LDFLAGS =-s -x -M         # GNU 连接器 gld 运行时用到的选项。含义是：-s 输出文件中省略所
                          # 有的符号信息；-x 删除所有局部符号；-M 表示需要在标准输出设备
                          # (显示器)上打印连接映象(link map)，是指由连接程序产生的一种
                          # 内存地址映象，其中列出了程序段装入到内存中的位置信息。具体
                          # 来讲有如下信息：
                          # • 目标文件及符号信息映射到内存中的位置；
                          # • 公共符号如何放置；
                          # • 连接中包含的所有文件成员及其引用的符号。
```

```
CC      =gcc $(RAMDISK)    # gcc 是 GNU C 程序编译器。对于 UNIX 类的脚本(script)程序而言，
# 在引用定义的标识符时，需在前面加上$符号并用括号括住标识符。
CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer \
-fcombine-regs -mstring-insns  # gcc 的选项。前一行最后的'\'符号表示下一行是续行。
                               # 选项含义为：-Wall 打印所有警告信息；-O 对代码进行优化；
                               # -fstrength-reduce 优化循环语句；-mstring-insns 是
                               # Linus 自己为 gcc 增加的选项，可以去掉。
CPP     =cpp -nostdinc -Iinclude  # cpp 是 gcc 的前(预)处理程序。-nostdinc -Iinclude 的含
                                  # 义是不要搜索标准的头文件目录中的文件，而是使用-I
                                  # 选项指定的目录或者是在当前目录里搜索头文件。

#
# ROOT_DEV specifies the default root-device when making the image.
# This can be either FLOPPY, /dev/xxxx or empty, in which case the
# default of /dev/hd6 is used by 'build'.
#
ROOT_DEV=/dev/hd6    # ROOT_DEV 指定在创建内核映像(image)文件时所使用的默认根文件系统所
                     # 在的设备，这可以是软盘(FLOPPY)、/dev/xxxx 或者干脆空着，空着时
                     # build 程序（在 tools/目录中）就使用默认值/dev/hd6。

ARCHIVES=kernel/kernel.o mm/mm.o fs/fs.o  # kernel 目录、mm 目录和 fs 目录所产生的目标代
                                          # 码文件。为了方便引用在这里将它们用
                                          # ARCHIVES（归档文件）标识符表示。
DRIVERS =kernel/blk_drv/blk_drv.a kernel/chr_drv/chr_drv.a  # 块和字符设备库文件。.a 表
                               # 示该文件是个归档文件，也即包含有许多可执行二进制代码子程
                               # 序集合的库文件，通常是用 GNU 的 ar 程序生成。ar 是 GNU 的二进制
                               # 文件处理程序，用于创建、修改以及从归档文件中抽取文件。
MATH    =kernel/math/math.a       # 数学运算库文件。
LIBS    =lib/lib.a                # 由 lib/目录中的文件所编译生成的通用库文件。

.c.s:                        # make 老式的隐式后缀规则。该行指示 make 利用下面的命令将所有的
                             # .c 文件编译生成.s 汇编程序。':'表示下面是该规则的命令。
        $(CC) $(CFLAGS) \
        -nostdinc -Iinclude -S -o $*.s $<  # 指使 gcc 采用前面 CFLAGS 所指定的选项以及
                             # 仅使用 include/目录中的头文件，在适当地编译后不进行汇编就
                             # 停止（-S），从而产生与输入的各个 C 文件对应的汇编语言形式的
                             # 代码文件。默认情况下所产生的汇编程序文件是原 C 文件名去掉.c
                             # 而加上.s 后缀。-o 表示其后是输出文件的形式。其中$*.s（或$@）
                             # 是自动目标变量，$<代表第一个先决条件，这里即是符合条件
                             # *.c 的文件。
.s.o:                          # 表示将所有.s 汇编程序文件编译成.o 目标文件。下一条是实
                               # 现该操作的具体命令。
        $(AS) -c -o $*.o $<    # 使用 gas 编译器将汇编程序编译成.o 目标文件。-c 表示只编译
                               # 或汇编，但不进行连接操作。
.c.o:                                    # 类似上面，*.c 文件-→*.o 目标文件。
        $(CC) $(CFLAGS) \
        -nostdinc -Iinclude -c -o $*.o $<  # 使用 gcc 将 C 语言文件编译成目标文件但不连接。

all:    Image              # all 表示创建 Makefile 所知的最顶层的目标。这里即是 image 文件。

Image: boot/bootsect boot/setup tools/system tools/build  # 说明目标（Image 文件）是由
                           # 分号后面的 4 个元素产生，分别是 boot/目录中的 bootsect 和
                           # setup 文件、tools/目录中的 system 和 build 文件。
```

```
42          tools/build boot/bootsect boot/setup tools/system $(ROOT_DEV) > Image
43          sync                  # 这两行是执行的命令。第一行表示使用 tools 目录下的 build 工具
                                  # 程序（下面会说明如何生成）将 bootsect、setup 和 system 文件
                                  # 以$(ROOT_DEV)为根文件系统设备组装成内核映像文件 Image。
                                  # 第二行的 sync 同步命令是迫使缓冲块数据立即写盘并更新超级块。
44
45 disk: Image                                        # 表示 disk 这个目标要由 Image 产生。
46          dd bs=8192 if=Image of=/dev/PS0   # dd 为 UNIX 标准命令：复制一个文件，根据选项
                                              # 进行转换和格式化。bs=表示一次读/写的字节数。
                                              # if=表示输入的文件，of=表示输出到的文件。
                                              # 这里/dev/PS0 是指第一个软盘驱动器(设备文件)。
47
48 tools/build: tools/build.c                 # 由 tools 目录下的 build.c 程序生成执行程序 build。
49          $(CC) $(CFLAGS) \
50          -o tools/build tools/build.c      # 编译生成执行程序 build 的命令。
51
52 boot/head.o: boot/head.s                   # 利用上面给出的.s.o 规则生成 head.o 目标文件。
53
54 tools/system:    boot/head.o init/main.o \
55                  $(ARCHIVES) $(DRIVERS) $(MATH) $(LIBS)   # 表示 tools 目录中的 system 文件
                                                             # 要由分号右边所列的元素生成。
56          $(LD) $(LDFLAGS) boot/head.o init/main.o \
57          $(ARCHIVES) \
58          $(DRIVERS) \
59          $(MATH) \
60          $(LIBS) \
61          -o tools/system > System.map      # 生成 system 的命令。最后的 > System.map 表示
                                              # gld 需要将连接映象重定向存放在 System.map 文件中。
                                              # 关于 System.map 文件的用途参见注释后的说明。
62
63 kernel/math/math.a:                        # 数学协处理函数文件 math.a 由下一行上的命令实现。
64          (cd kernel/math; make)            # 进入 kernel/math/目录；运行 make 工具程序。
                                              # 下面从 66--82 行的含义与此处的类似。
65
66 kernel/blk_drv/blk_drv.a:                  # 块设备函数文件 blk_drv.a
67          (cd kernel/blk_drv; make)
68
69 kernel/chr_drv/chr_drv.a:                  # 字符设备函数文件 chr_drv.a
70          (cd kernel/chr_drv; make)
71
72 kernel/kernel.o:                           # 内核目标模块 kernel.o
73          (cd kernel; make)
74
75 mm/mm.o:                                   # 内存管理模块 mm.o
76          (cd mm; make)
77
78 fs/fs.o:                                   # 文件系统目标模块 fs.o
79          (cd fs; make)
80
81 lib/lib.a:                                 # 库函数 lib.a
82          (cd lib; make)
83
84 boot/setup: boot/setup.s                             # 这里开始的三行是使用 8086 汇编和连接器
```

```
85          $(AS86) -o boot/setup.o boot/setup.s    # 对 setup.s 文件进行编译生成 setup 文件。
86          $(LD86) -s -o boot/setup boot/setup.o   # -s 选项表示要去除目标文件中的符号信息。
87
88  boot/bootsect:  boot/bootsect.s                         # 同上。生成 bootsect.o 磁盘引导块。
89          $(AS86) -o boot/bootsect.o boot/bootsect.s
90          $(LD86) -s -o boot/bootsect boot/bootsect.o
91
92  tmp.s:  boot/bootsect.s tools/system    # 从 92--95 这四行的作用是在 bootsect.s 程序开头添加
        # 一行有关 system 文件长度信息。方法是首先生成含有“SYSSIZE = system 文件实际长度”
        # 一行信息的 tmp.s 文件，然后将 bootsect.s 文件添加在其后。取得 system 长度的方法是：
        # 首先利用命令 ls 对 system 文件进行长列表显示，用 grep 命令取得列表行上文件字节数字段
        # 信息，并定向保存在 tmp.s 临时文件中。cut 命令用于剪切字符串，tr 用于去除行尾的回车符。
        # 其中：(实际长度 + 15)/16 用于获得用‘节’表示的长度信息。1 节 = 16 字节。
93          (echo -n "SYSSIZE = (";ls -l tools/system | grep system \
94                  | cut -c25-31 | tr '\012' ' '; echo "+ 15 ) / 16") > tmp.s
95          cat boot/bootsect.s >> tmp.s
96
97  clean:  # 当执行'make clean'时，就会执行 98--103 行上的命令，去除所有编译连接生成的文件。
            # 'rm'是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
98          rm -f Image System.map tmp_make core boot/bootsect boot/setup
99          rm -f init/*.o tools/system tools/build boot/*.o
100         (cd mm;make clean)          # 进入 mm/目录；执行该目录 Makefile 文件中的 clean 规则。
101         (cd fs;make clean)
102         (cd kernel;make clean)
103         (cd lib;make clean)
104
105 backup: clean   # 该规则将首先执行上面的 clean 规则，然后对 linux/目录进行压缩，生成
                # backup.Z 压缩文件。'cd .. '表示退到 linux/的上一级（父）目录；
                # 'tar cf - linux'表示对 linux/目录执行 tar 归档程序。-cf 表示需要创建
                # 新的归档文件 '| compress -'表示将 tar 程序的执行通过管道操作('|')
                # 传递给压缩程序 compress，并将压缩程序的输出存成 backup.Z 文件。
106         (cd .. ; tar cf - linux | compress - > backup.Z)
107         sync    # 迫使缓冲块数据立即写盘并更新磁盘超级块。
108
109 dep:
    # 该目标或规则用于各文件之间的依赖关系。创建的这些依赖关系是为了给 make 用来确定是否需要要
    # 重建一个目标对象的。比如当某个头文件被改动过后，make 就通过生成的依赖关系，重新编译与该
    # 头文件有关的所有*.c 文件。具体方法如下：
    # 使用字符串编辑程序 sed 对 Makefile 文件（这里即是自己）进行处理，输出为删除 Makefile
    # 文件中'### Dependencies'行后面的所有行（下面从 118 开始的行），并生成 tmp_make
    # 临时文件（也即 110 行的作用）。然后对 init/目录下的每一个 C 文件（其实只有一个文件
    # main.c）执行 gcc 预处理操作，-M 标志告诉预处理程序输出描述每个目标文件相关性的规则，
    # 并且这些规则符合 make 语法。对于每一个源文件，预处理程序输出一个 make 规则，其结果
    # 形式是相应源程序文件的目标文件名加上其依赖关系--该源文件中包含的所有头文件列表。
    # 111 行中的$$i 实际上是$($i)的意思。这里$i 是这句前面的 shell 变量的值。
    # 然后把预处理结果都添加到临时文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。
110         sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
111         (for i in init/*.c;do echo -n "init/";$(CPP) -M $$i;done) >> tmp_make
112         cp tmp_make Makefile
113         (cd fs; make dep)           # 对 fs/目录下的 Makefile 文件也作同样的处理。
114         (cd kernel; make dep)
115         (cd mm; make dep)
116
```

```
### Dependencies:
init/main.o : init/main.c include/unistd.h include/sys/stat.h \
  include/sys/types.h include/sys/times.h include/sys/utsname.h \
  include/utime.h include/time.h include/linux/tty.h include/termios.h \
  include/linux/sched.h include/linux/head.h include/linux/fs.h \
  include/linux/mm.h include/signal.h include/asm/system.h include/asm/io.h \
  include/stddef.h include/stdarg.h include/fcntl.h
```

2.7 其它信息

2.7.1 Makefile 简介

makefile 文件是 make 工具程序的配置文件。Make 工具程序的主要用途是能自动地决定一个含有很多源程序文件的大型程序中哪个文件需要被重新编译。makefile 的使用比较复杂，这里只是根据上面的 makefile 文件作些简单的介绍。详细说明请参考 GNU make 使用手册。

为了使用 make 程序，你就需要 makefile 文件来告诉 make 要做些什么工作。通常，makefile 文件会告诉 make 如何编译和连接一个文件。当明确指出时，makefile 还可以告诉 make 运行各种命令（例如，作为清理操作而删除某些文件）。

make 的执行过程分为两个不同的阶段。在第一个阶段，它读取所有的 makefile 文件以及包含的 makefile 文件等，记录所有的变量及其值、隐式的或显式的规则，并构造出所有目标对象及其先决条件的一幅全景图。在第二阶段期间，make 就使用这些内部结构来确定哪个目标对象需要被重建，并且使用相应的规则来操作。

当 make 重新编译程序时，每个修改过的 C 代码文件必须被重新编译。如果一个头文件被修改过了，那么为了确保正确，每一个包含该头文件的 C 代码程序都将被重新编译。每次编译操作都产生一个与源程序对应的目标文件(object file)。最终，如果任何源代码文件被编译过了，那么所有的目标文件不管是刚编译完的还是以前就编译好的必须连接在一起以生成新的可执行文件。

简单的 makefile 文件含有一些规则，这些规则具有如下的形式：

```
目标(target)...  : 先决条件(prerequisites)...
                命令(command)
                ...
                ...
```

其中’目标’对象通常是程序生成的一个文件的名称；例如是一个可执行文件或目标文件。目标也可以是所要采取活动的名字，比如’清除’(’clean’)。’先决条件’是一个或多个文件名，是用作产生目标的输入条件。通常一个目标依赖几个文件。而’命令’是 make 需要执行的操作。一个规则可以有多个命令，每一个命令自成一行。请注意，你需要在每个命令行之前键入一个制表符！这是粗心者常常忽略的地方。

如果一个先决条件通过目录搜寻而在另外一个目录中被找到，这并不会改变规则的命令；它们将被如期执行。因此，你必须小心地设置命令，使得命令能够在 make 发现先决条件的目录中找到需要的先决条件。这就需要通过使用自动变量来做到。自动变量是一种在命令行上根据具体情况能被自动替换的变量。自动变量的值是基于目标对象及其先决条件而在命令执行前设置的。例如，’$^’的值表示规则的所有先决条件，包括它们所处目录的名称；’$<’的值表示规则中的第一个先决条件；’$@’表示目标对象；另外还有一些自动变量这里就不提了。

有时，先决条件还常包含头文件，而这些头文件并不愿在命令中说明。此时自动变量’$<’正是第一个先决条件。例如：

```
foo.o : foo.c defs.h hack.h
        cc -c $(CFLAGS) $< -o $@
```

其中的’$<’就会被自动地替换成 foo.c，而$@则会被替换为 foo.o

为了让 make 能使用习惯用法来更新一个目标对象，你可以不指定命令，写一个不带命令的规则或者不写规则。此时 make 程序将会根据源程序文件的类型（程序的后缀）来判断要使用哪个隐式规则。

后缀规则是为 make 程序定义隐式规则的老式方法。(现在这种规则已经不用了，取而代之的是使用更通用更清晰的模式匹配规则)。下面例子就是一种双后缀规则。双后缀规则是用一对后缀定义的：源后缀和目标后缀。相应的隐式先决条件是通过使用文件名中的源后缀替换目标后缀后得到。因此，此时下面的’$<’值是*.c 文件名。而正条 make 规则的含义是将*.c 程序编译成*.s 代码。

```
.c.s:
        $(CC) $(CFLAGS) \
        -nostdinc -Iinclude -S -o $*.s $<
```

通常命令是属于一个具有先决条件的规则，并在任何先决条件改变时用于生成一个目标(target)文件。然而，为目标而指定命令的规则也并不一定要有先决条件。例如，与目标'clean'相关的含有删除(delete)命令的规则并不需要有先决条件。此时，一个规则说明了如何以及何时来重新制作某些文件，而这些文件是特定规则的目标。make 根据先决条件来执行命令以创建或更新目标。一个规则也可以说明如何及何时执行一个操作。

一个 makefile 文件也可以含有除规则以外的其它文字，但一个简单的 makefile 文件只需要含有适当的规则。规则可能看上去要比上面示出的模板复杂得多，但基本上都是符合的。

makefile 文件最后生成的依赖关系是用于让 make 来确定是否需要重建一个目标对象。比如当某个头文件被改动过后，make 就通过这些依赖关系，重新编译与该头文件有关的所有*.c 文件。

2.7.2 as86,ld86 简介

as86 和 ld86 是由 Bruce Evans 编写的 Intel 8086 汇编编译程序和连接程序。它完全是一个 8086 的汇编编译器，但却可以为 386 处理器编制 32 位的代码。Linux 使用它仅仅是为了创建 16 位的启动扇区(bootsector)代码和 setup 二进制执行代码。该编译器的语法与 GNU 的汇编编译器的语法是不兼容的，但近似于 Intel 的汇编语言语法（如操作数的次序相反等)。

Bruce Evans 是 minix 操作系统 32 位版本的主要编制者，他与 Linux 的创始人 Linus Torvalds 是很好的朋友。Linus 本人也从 Bruce Evans 那里学到了不少有关 UNIX 类操作系统的知识，minix 操作系统的不足之处也是两个好朋友互相探讨得出的结果，这激发了 Linus 在 Intel 386 体系结构上开发一个全新概念的操作系统，因此 Linux 操作系统的诞生与 Bruce Evans 也有着密切的关系。

有关这个编译器和连接器的源代码可以从 FTP 服务器 ftp.funet.fi 上或从我的网站(www.oldlinux.org)上下载。

这两个程序的使用方法和选项如下：

```
as 的使用方法和选项：
--------------------
as [-03agjuw] [-b [bin]] [-lm [list]] [-n name] [-o obj] [-s sym] src
```

默认设置 (除了以下默认值以外，其它选项默认为关闭或无；若没有明确说明 a 标志，则不会有输出):
-03 32 位输出；
list 在标准输出上显示；
name 源文件的基本名称（也即不包括“.“后的扩展名）；
选项含义：
-0 从 16 比特代码段开始；
-3 从 32 比特代码段开始；
-a 开启与 as、ld 的部分兼容性选项；
-b 产生二进制文件，后面可以跟文件名；
-g 在目标文件中仅存入全局符号；
-j 使所有跳转语句均为长跳转；
-l 产生列表文件，后面可以跟随列表文件名；
-m 在列表中扩展宏定义；
-n 后面跟随模块名称（取代源文件名称放入目标文件中）；
-o 产生目标文件，后跟目标文件名；
-s 产生符号文件，后跟符号文件名；
-u 将未定义符号作为输入的未指定段的符号；
-w 不显示警告信息；

ld 连接器的使用语法和选项：

对于生成 Minix a.out 格式的版本：
ld [-03Mims[-]] [-T textaddr] [-llib_extension] [-o outfile] infile...

对于生成 GNU-Minix 的 a.out 格式的版本：
ld [-03Mimrs[-]] [-T textaddr] [-llib_extension] [-o outfile] infile...

默认设置(除了以下默认值以外，其它选项默认为关闭或无):
-03 32 位输出；
outfile a.out 格式输出；

-0 产生具有 16 比特魔数的头结构，并且对-lx 选项使用 i86 子目录；
-3 产生具有 32 比特魔数的头结构，并且对-lx 选项使用 i386 子目录；
-M 在标准输出设备上显示已链接的符号；
-T 后面跟随文本基地址 (使用适合于 strtoul 的格式)；
-i 分离的指令与数据段（I&D）输出；
-lx 将库/local/lib/subdir/libx.a 加入链接的文件列表中；
-m 在标准输出设备上显示已链接的模块；
-o 指定输出文件名，后跟输出文件名；
-r 产生适合于进一步重定位的输出；
-s 在目标文件中删除所有符号。

2.7.3 System.map 文件

System.map 文件用于存放内核符号表信息。符号表是所有符号及其对应地址的一个列表。随着每次内核的编译，就会产生一个新的对应 System.map 文件。当内核运行出错时，通过 System.map 文件中的符号表解析，就可以查到一个地址值对应的变量名，或反之。

利用 System.map 符号表文件，在内核或相关程序出错时，就可以获得我们比较容易识别的信息。符号表的样例如下所示：

```
c03441a0 B dmi_broken
c03441a4 B is_sony_vaio_laptop
```

```
c03441c0 b dmi_ident
c0344200 b pci_bios_present
c0344204 b pirq_table
```

可以看出名称为 dmi_broken 的变量位于内核地址 c03441a0 处。

System.map 位于使用它的软件(例如内核日志记录后台程序 klogd)能够寻找到的地方。在系统启动时，如果没有以一个参数的形式为 klogd 给出 System.map 的位置，则 klogd 将会在三个地方搜寻 System.map。依次为：

1. /boot/System.map
2. /System.map
3. /usr/src/linux/System.map

尽管内核本身实际上不使用 System.map，但其它程序，象 klogd，lsof，ps 以及其它许多软件，象 dosemu，都需要有一个正确的 System.map 文件。利用该文件，这些程序就可以根据已知的内存地址查找出对应的内核变量名称，便于对内核的调试工作。

2.8 本章小结

本章概述了 Linux 早期操作系统的内核模式和体系结构。给出了 Linux 0.11 内核源代码的目录结构形式，并详细地介绍了各个子目录中代码文件的基本功能和层次关系。然后介绍了在 RedHat 9 系统下编译 linux 0.11 内核时，对代码需要进行修改的地方。最后从 linux 内核主目录下的 makefile 文件着手，开始对内核源代码进行注释。

第 3 章 引导启动程序（boot）

这里先总的说明一下 Linux 操作系统启动部分的主要执行流程。当 PC 的电源打开后，80x86 结构的 CPU 将自动进入实模式，并从地址 0xFFFF0 开始自动执行程序代码，这个地址通常是 ROM-BIOS 中的地址。PC 机的 BIOS 将执行某些系统的检测，并在物理地址 0 处开始初始化中断向量。此后，它将可启动设备的第一个扇区（磁盘引导扇区，512 字节）读入内存绝对地址 0x7C00 处，并跳转到这个地方。启动设备通常是软驱或是硬盘。这里的叙述是非常简单的，但这已经足够理解内核初始化的工作过程了。

Linux 的最最前面部分是用 8086 汇编语言编写的(boot/bootsect.s)，它将由 BIOS 读入到内存绝对地址 0x7C00(31KB)处，当它被执行时就会把自己移到绝对地址 0x90000(576KB)处，并把启动设备中后 2kB 字节代码(boot/setup.s)读入到内存 0x90200 处，而内核的其它部分（system 模块）则被读入到从地址 0x10000 开始处，因为当时 system 模块的长度不会超过 0x80000 字节大小（即 512KB），所以它不会覆盖在 0x90000 处开始的 bootsect 和 setup 模块。随后将 system 模块移动到内存起始处，这样 system 模块中代码的地址也即等于实际的物理地址。便于对内核代码和数据的操作。图 3.1 清晰地显示出 Linux 系统启动时这几个程序或模块在内存中的动态位置。其中，每一竖条框代表某一时刻内存中各程序的映像位置图。在系统加载期间将显示信息"Loading..."。然后控制权将传递给 boot/setup.s 中的代码，这是另一个实模式汇编语言程序。

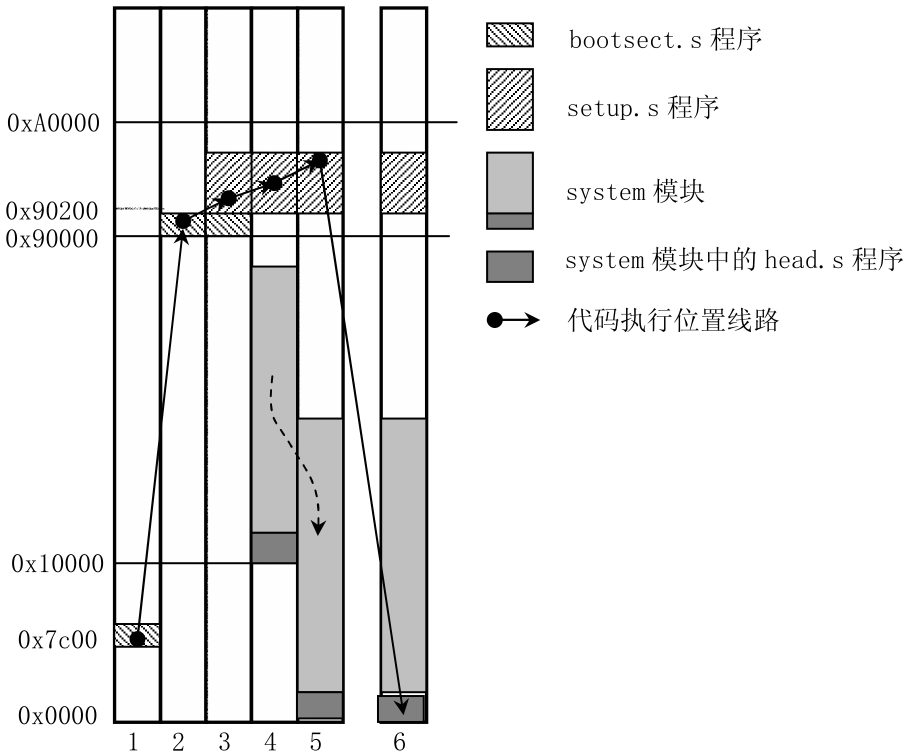


图 3.1 启动引导时内核在内存中的位置和移动后的位置情况

启动部分识别主机的某些特性以及 vga 卡的类型。如果需要，它会要求用户为控制台选择显示模式。然后将整个系统从地址 0x10000 移至 0x0000 处，进入保护模式并跳转至系统的余下部分(在 0x0000 处)。此时所有 32 位运行方式的设置启动被完成: IDT、GDT 以及 LDT 被加载，处理器和协处理器也已确认，

分页工作也设置好了；最终调用 init/main.c 中的 main()程序。上述操作的源代码是在 boot/head.S 中的，这可能是整个内核中最有诀窍的代码了。注意如果在前述任何一步中出了错，计算机就会死锁。在操作系统还没有完全运转之前是处理不了出错的。

3.1 概述

本章主要描述 boot/目录中的三个汇编代码文件，见列表 2.1 所示。正如在前一章中提到的，这三个文件虽然都是汇编程序，但却使用了两种语法格式。Bootsect.s 和 setup.s 采用近似于 Intel 的汇编语言语法，需要使用 Intel 8086 汇编编译器和连接器 as86 和 ld86，而 head.s 则使用 GNU 的汇编程序格式，需要用 GNU 的 as 进行编译。这是一种 AT&T 语法的汇编语言程序。

列表 3.1 linux/boot/目录

文件名	长度(字节)	最后修改时间(GMT)	说明
bootsect.s	5052 bytes	1991-12-05 22:47:58	
head.s	5938 bytes	1991-11-18 15:05:09	
setup.s	5364 bytes	1991-12-05 22:48:10	

阅读这些代码除了你需要知道一些一般 8086 汇编语言的知识以外，还要对采用 Intel 80X86 微处理器的 PC 机的体系结构以及 80386 32 位保护模式下的编程原理有些了解。所以在开始阅读源代码之前可以先大概浏览一下附录中有关 PC 机硬件接口控制编程和 80386 32 位保护模式的编程方法，在阅读代码时再就事论事地针对具体问题参考附录中的详细说明。

3.2 bootsect.s 文件

3.2.1 功能描述

bootsect.s 代码是磁盘引导块程序，驻留在磁盘的第一个扇区中（引导扇区，0 磁道（柱面），0 磁头，第 1 个扇区）。在 PC 机加电 ROM BIOS 自检后，引导扇区由 BIOS 加载到内存 0x7C00 处，然后将自己移动到内存 0x90000 处。该程序的主要作用是首先将 setup 模块（由 setup.s 编译成）从磁盘加载到内存，紧接着 bootsect 的后面位置（0x90200），然后利用 BIOS 中断 0x13 取磁盘参数表中当前启动引导盘的参数，接着在屏幕上显示“Loading system...”字符串。再者将 system 模块从磁盘上加载到内存 0x10000 开始的地方。随后确定根文件系统的设备号，若没有指定，则根据所保存的引导盘的每磁道扇区数判别出盘的类型和种类（是 1.44M A 盘？）并保存其设备号于 root_dev(引导块的 0x508 地址处)，最后长跳转到 setup 程序的开始处（0x90200）执行 setup 程序。

3.2.2 代码注释

列表 3.2 linux/boot/bootsect.s 程序

```
!
! SYS_SIZE is the number of clicks (16 bytes) to be loaded.
```

```
! 0x3000 is 0x30000 bytes = 196kB, more than enough for current
! versions of linux   ! SYS_SIZE 是要加载的节数（16 字节为 1 节）。0x3000 共为
                      ! 0x30000 字节=192 kB（上面 Linus 估算错了），对于当前的版本空间已足够了。
!
SYSSIZE = 0x3000      ! 指编译连接后 system 模块的大小。参见列表 1.2 中第 92 的说明。
                      ! 这里给出了一个最大默认值。
!
!      bootsect.s              (C) 1991 Linus Torvalds
!
! bootsect.s is loaded at 0x7c00 by the bios-startup routines, and moves
! iself out of the way to address 0x90000, and jumps there.
!
! It then loads 'setup' directly after itself (0x90200), and the system
! at 0x10000, using BIOS interrupts.
!
! NOTE! currently system is at most 8*65536 bytes long. This should be no
! problem, even in the future. I want to keep it simple. This 512 kB
! kernel size should be enough, especially as this doesn't contain the
! buffer cache as in minix
!
! The loader has been made as simple as possible, and continuos
! read errors will result in a unbreakable loop. Reboot by hand. It
! loads pretty fast by getting whole sectors at a time whenever possible.
!
! 以下是前面这些文字的翻译：
!     bootsect.s              (C) 1991 Linus Torvalds 版权所有
!
! bootsect.s 被 bios-启动子程序加载至 0x7c00 (31k)处，并将自己
! 移到了地址 0x90000 (576k)处，并跳转至那里。
!
! 它然后使用 BIOS 中断将'setup'直接加载到自己的后面(0x90200)(576.5k)，
! 并将 system 加载到地址 0x10000 处。
!
! 注意! 目前的内核系统最大长度限制为(8*65536)(512k)字节，即使是在
! 将来这也应该没有问题的。我想让它保持简单明了。这样 512k 的最大内核长度应该
! 足够了，尤其是这里没有象 minix 中一样包含缓冲区高速缓冲。
!
! 加载程序已经做的够简单了，所以持续的读出错将导致死循环。只能手工重启。
! 只要可能，通过一次取取所有的扇区，加载过程可以做的很快的。

.globl begtext, begdata, begbss, endtext, enddata, endbss  ! 定义了 6 个全局标识符；
.text                         ! 文本段；
begtext:
.data                         ! 数据段；
begdata:
.bss                          ! 堆栈段；
begbss:
.text                         ! 文本段；

SETUPLEN = 4                            ! nr of setup-sectors
                                        ! setup 程序的扇区数(setup-sectors)值；
BOOTSEG  = 0x07c0                       ! original address of boot-sector
                                        ! bootsect 的原始地址（是段地址，以下同）；
```

```
36 INITSEG  = 0x9000                        ! we move boot here - out of the way
                                            ! 将 bootsect 移到这里 -- 避开;
37 SETUPSEG = 0x9020                        ! setup starts here
                                            ! setup 程序从这里开始;
38 SYSSEG   = 0x1000                        ! system loaded at 0x10000 (65536).
                                            ! system 模块加载到 0x10000 (64 kB) 处;
39 ENDSEG   = SYSSEG + SYSSIZE              ! where to stop loading
                                            ! 停止加载的段地址;
40
41 ! ROOT_DEV:     0x000 - same type of floppy as boot.
   !                       根文件系统设备使用与引导时同样的软驱设备;
42 !               0x301 - first partition on first drive etc
   !                       根文件系统设备在第一个硬盘的第一个分区上，等等;
43 ROOT_DEV = 0x306    ! 指定根文件系统设备是第 2 个硬盘的第 1 个分区。这是 Linux 老式的硬盘命名
                       ! 方式,具体值的含义如下:
                       ! 设备号=主设备号*256 + 次设备号（也即 dev_no = (major<<8) + minor ）
                       ! （主设备号：1-内存,2-磁盘,3-硬盘,4-ttyx,5-tty,6-并行口,7-非命名管道）
                       ! 0x300 - /dev/hd0 - 代表整个第 1 个硬盘;
                       ! 0x301 - /dev/hd1 - 第 1 个盘的第 1 个分区;
                       !  …
                       ! 0x304 - /dev/hd4 - 第 1 个盘的第 4 个分区;
                       ! 0x305 - /dev/hd5 - 代表整个第 2 个硬盘盘;
                       ! 0x306 - /dev/hd6 - 第 2 个盘的第 1 个分区;
                       !  …
                       ! 0x309 - /dev/hd9 - 第 2 个盘的第 4 个分区;
                       ! 从 linux 内核 0.95 版后已经使用与现在相同的命名方法了。
44
45 entry start         ! 告知连接程序，程序从 start 标号开始执行。
46 start:                        ! 47--56 行作用是将自身(bootsect)从目前段位置 0x07c0(31k)
                                 ! 移动到 0x9000(576k)处，共 256 字（512 字节），然后跳转到
                                 ! 移动后代码的 go 标号处，也即本程序的下一语句处。
47         mov     ax,#BOOTSEG   ! 将 ds 段寄存器置为 0x7C0;
48         mov     ds,ax
49         mov     ax,#INITSEG   ! 将 es 段寄存器置为 0x9000;
50         mov     es,ax
51         mov     cx,#256       ! 移动计数值=256 字;
52         sub     si,si         ! 源地址   ds:si = 0x07C0:0x0000
53         sub     di,di         ! 目的地址 es:di = 0x9000:0x0000
54         rep                   ! 重复执行，直到 cx = 0
55         movw                  ! 移动 1 个字;
56         jmpi    go,INITSEG    ! 间接跳转。这里 INITSEG 指出跳转到的段地址。
57 go:     mov     ax,cs         ! 将 ds、es 和 ss 都置成移动后代码所在的段处(0x9000)。
58         mov     ds,ax         ! 由于程序中有堆栈操作(push,pop,call)，因此必须设置堆栈。
59         mov     es,ax
60 ! put stack at 0x9ff00.       ! 将堆栈指针 sp 指向 0x9ff00(即 0x9000:0xff00)处
61         mov     ss,ax
62         mov     sp,#0xFF00           ! arbitrary value >>512
                                 ! 由于代码段移动过了，所以要重新设置堆栈段的位置。
                                 ! sp 只要指向远大于 512 偏移（即地址 0x90200）处
                                 ! 都可以。因为从 0x90200 地址开始处还要放置 setup 程序，
                                 ! 而此时 setup 程序大约为 4 个扇区，因此 sp 要指向大
                                 ! 于（200 + 200 * 4 + 堆栈大小）处。
63
```

```
64 ! load the setup-sectors directly after the bootblock.
65 ! Note that 'es' is already set up.
   ! 在 bootsect 程序块后紧根着加载 setup 模块的代码数据。
   ! 注意 es 已经设置好了。（在移动代码时 es 已经指向目的段地址处 0x9000）。
66
67 load_setup:
     ! 68--77 行的用途是利用 BIOS 中断 INT 0x13 将 setup 模块从磁盘第 2 个扇区
     ! 开始读到 0x90200 开始处，共读 4 个扇区。如果读出错，则复位驱动器，并
     ! 重试，没有退路。INT 0x13 的使用方法如下：
     ! 读扇区：
     ! ah = 0x02 - 读磁盘扇区到内存；al = 需要读出的扇区数量；
     ! ch = 磁道(柱面)号的低 8 位；   cl = 开始扇区(0-5 位)，磁道号高 2 位(6-7)；
     ! dh = 磁头号；                  dl = 驱动器号（如果是硬盘则要置位 7）；
     ! es:bx →指向数据缓冲区；   如果出错则 CF 标志置位。
68         mov     dx,#0x0000              ! drive 0, head 0
69         mov     cx,#0x0002              ! sector 2, track 0
70         mov     bx,#0x0200              ! address = 512, in INITSEG
71         mov     ax,#0x0200+SETUPLEN     ! service 2, nr of sectors
72         int     0x13                    ! read it
73         jnc     ok_load_setup           ! ok - continue
74         mov     dx,#0x0000
75         mov     ax,#0x0000              ! reset the diskette
76         int     0x13
77         j       load_setup
78
79 ok_load_setup:
80
81 ! Get disk drive parameters, specifically nr of sectors/track
   ! 取磁盘驱动器的参数，特别是每道的扇区数量。
   ! 取磁盘驱动器参数 INT 0x13 调用格式和返回信息如下：
   ! ah = 0x08    dl = 驱动器号（如果是硬盘则要置位 7 为 1）。
   ! 返回信息：
   ! 如果出错则 CF 置位，并且 ah = 状态码。
   ! ah = 0，  al = 0，         bl = 驱动器类型（AT/PS2）
   ! ch = 最大磁道号的低 8 位，cl = 每磁道最大扇区数(位 0-5)，最大磁道号高 2 位(位 6-7)
   ! dh = 最大磁头数，          dl = 驱动器数量，
   ! es:di --→ 软驱磁盘参数表。
82
83         mov     dl,#0x00
84         mov     ax,#0x0800              ! AH=8 is get drive parameters
85         int     0x13
86         mov     ch,#0x00
87         seg cs                      ! 表示下一条语句的操作数在 cs 段寄存器所指的段中。
88         mov     sectors,cx          ! 保存每磁道扇区数。
89         mov     ax,#INITSEG
90         mov     es,ax               ! 因为上面取磁盘参数中断改掉了 es 的值，这里重新改回。
91
92 ! Print some inane message  ! 在显示一些信息('Loading system ...'回车换行，共 24 个字符)。
93
94         mov     ah,#0x03                ! read cursor pos
95         xor     bh,bh                   ! 读光标位置。
96         int     0x10
97
```

```
98          mov     cx,#24                  ! 共 24 个字符。
99          mov     bx,#0x0007              ! page 0, attribute 7 (normal)
100         mov     bp,#msg1                ! 指向要显示的字符串。
101         mov     ax,#0x1301              ! write string, move cursor
102         int     0x10                    ! 写字符串并移动光标。
103
104 ! ok, we've written the message, now
105 ! we want to load the system (at 0x10000)  ! 现在开始将 system 模块加载到 0x10000(64k)处。
106
107         mov     ax,#SYSSEG
108         mov     es,ax           ! segment of 0x010000   ! es = 存放 system 的段地址。
109         call    read_it         ! 读磁盘上 system 模块，es 为输入参数。
110         call    kill_motor      ! 关闭驱动器马达，这样就可以知道驱动器的状态了。
111
112 ! After that we check which root-device to use. If the device is
113 ! defined (!= 0), nothing is done and the given device is used.
114 ! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
115 ! on the number of sectors that the BIOS reports currently.
    ! 此后，我们检查要使用哪个根文件系统设备（简称根设备）。如果已经指定了设备(!=0)
    ! 就直接接使用给定的设备。否则就需要根据 BIOS 报告的每磁道扇区数来
    ! 确定到底使用/dev/PS0 (2,28) 还是 /dev/at0 (2,8)。
          ! 上面一行中两个设备文件的含义：
          ! 在 Linux 中软驱的主设备号是 2(参见第 43 行的注释)，次设备号 = type*4 + nr，其中
          ! nr 为 0-3 分别对应软驱 A、B、C 或 D；type 是软驱的类型（2→1.2M 或 7→1.44M 等）。
          ! 因为 7*4 + 0 = 28，所以 /dev/PS0 (2,28)指的是 1.44M A 驱动器,其设备号是 0x021c
          ! 同理 /dev/at0 (2,8)指的是 1.2M A 驱动器，其设备号是 0x0208。
116
117         seg cs
118         mov     ax,root_dev         ! 将根设备号
119         cmp     ax,#0
120         jne     root_defined
121         seg cs
122         mov     bx,sectors          ! 取上面第 88 行保存的每磁道扇区数。如果 sectors=15
                                        ! 则说明是 1.2Mb 的驱动器；如果 sectors=18，则说明是
                                        ! 1.44Mb 软驱。因为是可引导的驱动器，所以肯定是 A 驱。
123         mov     ax,#0x0208              ! /dev/ps0 - 1.2Mb
124         cmp     bx,#15              ! 判断每磁道扇区数是否=15
125         je      root_defined        ! 如果等于，则 ax 中就是引导驱动器的设备号。
126         mov     ax,#0x021c              ! /dev/PS0 - 1.44Mb
127         cmp     bx,#18
128         je      root_defined
129 undef_root:                             ! 如果都不一样，则死循环（死机）。
130         jmp undef_root
131 root_defined:
132         seg cs
133         mov     root_dev,ax         ! 将检查过的设备号保存起来。
134
135 ! after that (everyting loaded), we jump to
136 ! the setup-routine loaded directly after
137 ! the bootblock:
    ! 到此，所有程序都加载完毕，我们就跳转到被
    ! 加载在 bootsect 后面的 setup 程序去。
138
```

```
        jmpi    0,SETUPSEG          ! 跳转到 0x9020:0000(setup.s 程序的开始处)。
                                    !!!! 本程序到此就结束了。!!!!
! 下面是两个子程序。

! This routine loads the system at address 0x10000, making sure
! no 64kB boundaries are crossed. We try to load it as fast as
! possible, loading whole tracks whenever we can.
!
! in:   es - starting address segment (normally 0x1000)
!
! 该子程序将系统模块加载到内存地址 0x10000 处，并确定没有跨越 64KB 的内存边界。我们试图尽快
! 地进行加载，只要可能，就每次加载整条磁道的数据。
! 输入：es – 开始内存地址段值（通常是 0x1000）
sread:  .word 1+SETUPLEN           ! sectors read of current track
                                   ! 当前磁道中已读的扇区数。开始时已经读进 1 扇区的引导扇区
                                   ! bootsect 和 setup 程序所占的扇区数 SETUPLEN。
head:   .word 0                    ! current head   !当前磁头号。
track:  .word 0                    ! current track  !当前磁道号。

read_it:
! 测试输入的段值。必须位于内存地址 64KB 边界处，否则进入死循环。清 bx 寄存器，用于表示当前段
内
! 存放数据的开始位置。
        mov ax,es
        test ax,#0x0fff
die:    jne die                    ! es must be at 64kB boundary ! es 值必须位于 64KB 地址边界!
        xor bx,bx                  ! bx is starting address within segment ! bx 为段内偏移位置。
rp_read:
! 判断是否已经读入全部数据。比较当前所读段是否就是系统数据末端所处的段(#ENDSEG)，如果不是就
! 跳转至下面 ok1_read 标号处继续读数据。否则退出子程序返回。
        mov ax,es
        cmp ax,#ENDSEG             ! have we loaded all yet? ! 是否已经加载了全部数据？
        jb ok1_read
        ret
ok1_read:
! 计算和验证当前磁道需要读取的扇区数，放在 ax 寄存器中。
! 根据当前磁道还未读取的扇区数以及段内数据字节开始偏移位置，计算如果全部读取这些未读扇区，
所
! 读总字节数是否会超过 64KB 段长度的限制。若会超过，则根据此次最多能读入的字节数(64KB – 段
内
! 偏移位置)，反算出此次需要读取的扇区数。
        seg cs
        mov ax,sectors             ! 取每磁道扇区数。
        sub ax,sread               ! 减去当前磁道已读扇区数。
        mov cx,ax                  ! cx = ax = 当前磁道未读扇区数。
        shl cx,#9                  ! cx = cx * 512 字节。
        add cx,bx                  ! cx = cx + 段内当前偏移值(bx)
                                   !    = 此次读操作后，段内共读入的字节数。
        jnc ok2_read               ! 若没有超过 64KB 字节，则跳转至 ok2_read 处执行。
        je ok2_read
        xor ax,ax                  ! 若加上此次将读磁道上所有未读扇区时会超过 64KB，则计算
        sub ax,bx                  ! 此时最多能读入的字节数(64KB – 段内读偏移位置)，再转换
        shr ax,#9                  ! 成需要读取的扇区数。
```

```
173 ok2_read:
174         call read_track
175         mov cx,ax               ! cx = 该次操作已读取的扇区数。
176         add ax,sread            ! 当前磁道上已经读取的扇区数。
177         seg cs
178         cmp ax,sectors          ! 如果当前磁道上的还有扇区未读，则跳转到 ok3_read 处。
179         jne ok3_read
    ! 读该磁道的下一磁头面(1 号磁头)上的数据。如果已经完成，则去读下一磁道。
180         mov ax,#1
181         sub ax,head             ! 判断当前磁头号。
182         jne ok4_read            ! 如果是 0 磁头，则再去读 1 磁头面上的扇区数据。
183         inc track               ! 否则去读下一磁道。
184 ok4_read:
185         mov head,ax             ! 保存当前磁头号。
186         xor ax,ax               ! 清当前磁道已读扇区数。
187 ok3_read:
188         mov sread,ax            ! 保存当前磁道已读扇区数。
189         shl cx,#9               ! 上次已读扇区数*512 字节。
190         add bx,cx               ! 调整当前段内数据开始位置。
191         jnc rp_read            ! 若小于 64KB 边界值，则跳转到 rp_read(156 行)处，继续读数据。
                                    ! 否则调整当前段，为读下一段数据作准备。
192         mov ax,es
193         add ax,#0x1000          ! 将段基址调整为指向下一个 64KB 段内存。
194         mov es,ax
195         xor bx,bx               ! 清段内数据开始偏移值。
196         jmp rp_read             ! 跳转至 rp_read(156 行)处，继续读数据。
197
    ! 读当前磁道上指定开始扇区和需读扇区数的数据到 es:bx 开始处。参见第 67 行下对 BIOS 磁盘读中断
    ! int 0x13，ah=2 的说明。
    ! al - 需读扇区数；es:bx - 缓冲区开始位置。
198 read_track:
199         push ax
200         push bx
201         push cx
202         push dx
203         mov dx,track          ! 取当前磁道号。
204         mov cx,sread          ! 取当前磁道上已读扇区数。
205         inc cx                ! cl = 开始读扇区。
206         mov ch,dl             ! ch = 当前磁道号。
207         mov dx,head           ! 取当前磁头号。
208         mov dh,dl             ! dh = 磁头号。
209         mov dl,#0             ! dl = 驱动器号(为 0 表示当前驱动器)。
210         and dx,#0x0100        ! 磁头号不大于 1。
211         mov ah,#2             ! ah = 2，读磁盘扇区功能号。
212         int 0x13
213         jc bad_rt             ! 若出错，则跳转至 bad_rt。
214         pop dx
215         pop cx
216         pop bx
217         pop ax
218         ret
    ! 执行驱动器复位操作（磁盘中断功能号 0），再跳转到 read_track 处重试。
219 bad_rt: mov ax,#0
```

```
        mov dx,#0
        int 0x13
        pop dx
        pop cx
        pop bx
        pop ax
        jmp read_track

/*
 * This procedure turns off the floppy drive motor, so
 * that we enter the kernel in a known state, and
 * don't have to worry about it later.
 */
! 这个子程序用于关闭软驱的马达，这样我们进入内核后它处于已知状态，以后也就无须担心它了。
kill_motor:
        push dx
        mov dx,#0x3f2         ! 软驱控制卡的驱动端口，只写。
        mov al,#0             ! A 驱动器，关闭 FDC，禁止 DMA 和中断请求，关闭马达。
        outb                  ! 将 al 中的内容输出到 dx 指定的端口去。
        pop dx
        ret

sectors:
        .word 0                         ! 存放当前启动软盘每磁道的扇区数。

msg1:
        .byte 13,10                     ! 回车、换行的 ASCII 码。
        .ascii "Loading system ..."
        .byte 13,10,13,10               ! 共 24 个 ASCII 码字符。

.org 508                                ! 表示下面语句从地址 508(0x1FC)开始，所以 root_dev
                                        ! 在启动扇区的第 508 开始的 2 个字节中。
root_dev:
        .word ROOT_DEV                  ! 这里存放根文件系统所在的设备号(init/main.c 中会用)。
boot_flag:
        .word 0xAA55                    ! 硬盘有效标识。

.text
endtext:
.data
enddata:
.bss
endbss:
```

3.2.3 其它信息

对 bootsect.s 这段程序的说明和描述，在互连网上可以搜索到大量的资料。其中 Alessandro Rubini 著而由本人翻译的《Linux 内核源代码漫游》一篇文章比较详细地描述了内核启动的详细过程，很有参考价值。由于这段程序是在 386 实模式下运行的，因此相对来将比较容易理解。若此时阅读仍有困难，那么建议你首先再复习一下 80x86 汇编及其硬件的相关知识（可参阅参考文献[1]和[16]），然后再继续阅读本书。

对于最新开发的 Linux 内核，这段程序的改动也很小，基本保持了与 0.11 版的模样。

3.3 setup.s 程序

3.3.1 功能描述

setup 程序的作用主要是利用 ROM BIOS 中断读取机器系统数据，并将这些数据保存到 0x90000 开始的位置（覆盖掉了 bootsect 程序所在的地方），所取得的参数和保留的内存位置见下表 2.1 所示。这些参数将被内核中相关程序使用，例如字符设备驱动程序集中的 ttyio.c 程序等。

表 3.1 setup 程序读取并保留的参数

内存地址	长度(字节)	名称	描述
0x90000	2	光标位置	列号（0x00-最左端），行号（0x00-最顶端）
0x90002	2	扩展内存数	系统从 1M 开始的扩展内存数值（KB）。
0x90004	2	显示页面	当前显示页面
0x90006	1	显示模式	
0x90007	1	字符列数	
0x90008	2	??	
0x9000A	1	显示内存	显示内存(0x00-64k,0x01-128k,0x02-192k,0x03=256k)
0x9000B	1	显示状态	0x00-彩色,I/O=0x3dX；0x11-单色,I/O=0x3bX
0x9000C	2	特性参数	显示卡特性参数
...			
0x90080	16	硬盘参数表	第 1 个硬盘的参数表
0x90090	16	硬盘参数表	第 2 个硬盘的参数表（如果没有，则清零）
0x901FC	2	根设备号	根文件系统所在的设备号（bootsec.s 中设置）

然后 setup 程序将 system 模块从 0x10000-0x8ffff（当时认为内核系统模块 system 的长度不会超过此值：512KB）整块向下移动到内存绝对地址 0x00000 处。接下来加载中断描述符表寄存器(idtr)和全局描述符表寄存器(gdtr)，开启 A20 地址线，重新设置两个中断控制芯片 8259A，将硬件中断号重新设置为 0x20 - 0x2f。最后设置 CPU 的控制寄存器 CR0（也称机器状态字），从而进入 32 位保护模式运行，并跳转到位于 system 模块最前面部分的 head.s 程序继续运行。

3.3.2 代码注释

列表 3.3 linux/boot/setup.s 程序

```
!
!       setup.s         (C) 1991 Linus Torvalds
!
! setup.s is responsible for getting the system data from the BIOS,
! and putting them into the appropriate places in system memory.
! both setup.s and system has been loaded by the bootblock.
!
! This code asks the bios for memory/disk/other parameters, and
! puts them in a "safe" place: 0x90000-0x901FF, ie where the
! boot-block used to be. It is then up to the protected mode
! system to read them from there before the area is overwritten
! for buffer-blocks.
```

```
!
! setup.s 负责从 BIOS 中获取系统数据，并将这些数据放到系统内存的适当地方。
! 此时 setup.s 和 system 已经由 bootsect 引导块加载到内存中。
!
! 这段代码询问 bios 有关内存/磁盘/其它参数，并将这些参数放到一个
! “安全的”地方：0x90000-0x901FF，也即原来 bootsect 代码块曾经在
! 的地方，然后在被缓冲块覆盖掉之前由保护模式的 system 读取。
!

! NOTE! These had better be the same as in bootsect.s!
! 以下这些参数最好和 bootsect.s 中的相同！

INITSEG  = 0x9000        ! we move boot here - out of the way  ! 原来 bootsect 所处的段。
SYSSEG   = 0x1000        ! system loaded at 0x10000 (65536).  ! system 在 0x10000(64k)处。
SETUPSEG = 0x9020        ! this is the current segment  ! 本程序所在的段地址。

.globl begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text

entry start
start:

! ok, the read went well so we get current cursor position and save it for
! posterity.
! ok，整个读磁盘过程都正常，现在将光标位置保存以备今后使用。

        mov     ax,#INITSEG     ! this is done in bootsect already, but...
                                ! 将 ds 置成#INITSEG(0x9000)。这已经在 bootsect 程序中
                                ! 设置过，但是现在是 setup 程序，Linus 觉得需要再重新
                                ! 设置一下。
        mov     ds,ax
        mov     ah,#0x03        ! read cursor pos
                                ! BIOS 中断 0x10 的读光标功能号 ah = 0x03
                                ! 输入：bh = 页号
                                ! 返回：ch = 扫描开始线，cl = 扫描结束线，
                                ! dh = 行号(0x00 是顶端)，dl = 列号(0x00 是左边)。
        xor     bh,bh
        int     0x10            ! save it in known place, con_init fetches
        mov     [0],dx          ! it from 0x90000.
                                ! 上两句是说将光标位置信息存放在 0x90000 处，控制台
                                ! 初始化时会来取。

! Get memory size (extended mem, kB)  ! 下面 3 句取扩展内存的大小值（KB）。
                                ! 是调用中断 0x15，功能号 ah = 0x88
                                ! 返回：ax = 从 0x100000（1M）处开始的扩展内存大小(KB)。
                                ! 若出错则 CF 置位，ax = 出错码。

```

```
        mov     ah,#0x88
        int     0x15
        mov     [2],ax          ! 将扩展内存数值存在 0x90002 处（1 个字）。

! Get video-card data:  ! 下面这段用于取显示卡当前显示模式。
                        ! 调用 BIOS 中断 0x10，功能号 ah = 0x0f
                        ! 返回：ah = 字符列数，al = 显示模式，bh = 当前显示页。
                        ! 0x90004(1 字)存放当前页，0x90006 显示模式，0x90007 字符列数。

        mov     ah,#0x0f
        int     0x10
        mov     [4],bx          ! bh = display page
        mov     [6],ax          ! al = video mode, ah = window width

! check for EGA/VGA and some config parameters  ! 检查显示方式（EGA/VGA）并取参数。
                        ! 调用 BIOS 中断 0x10，附加功能选择 -取方式信息
                        ! 功能号：ah = 0x12，bl = 0x10
                        ! 返回：bh = 显示状态
                        !       (0x00 - 彩色模式，I/O 端口=0x3dX)
                        !       (0x01 - 单色模式，I/O 端口=0x3bX)
                        ! bl = 安装的显示内存
                        ! (0x00 - 64k, 0x01 - 128k, 0x02 - 192k, 0x03 = 256k)
                        ! cx = 显示卡特性参数(参见程序后的说明)。

        mov     ah,#0x12
        mov     bl,#0x10
        int     0x10
        mov     [8],ax    ! 0x90008 = ??
        mov     [10],bx   ! 0x9000A = 安装的显示内存，0x9000B = 显示状态(彩色/单色)
        mov     [12],cx   ! 0x9000C = 显示卡特性参数。

! Get hd0 data      ! 取第一个硬盘的信息（复制硬盘参数表）。
                    ! 第 1 个硬盘参数表的首地址竟然是中断向量 0x41 的向量值！而第 2 个硬盘
                    ! 参数表紧接第 1 个表的后面，中断向量 0x46 的向量值也指向这第 2 个硬盘
                    ! 的参数表首址。表的长度是 16 个字节(0x10)。
                    ! 下面两段程序分别复制 BIOS 有关两个硬盘的参数表，0x90080 处存放第 1 个
                    ! 硬盘的表，0x90090 处存放第 2 个硬盘的表。

        mov     ax,#0x0000
        mov     ds,ax
        lds     si,[4*0x41]   ! 取中断向量 0x41 的值，也即 hd0 参数表的地址➔ds:si
        mov     ax,#INITSEG
        mov     es,ax
        mov     di,#0x0080    ! 传输的目的地址: 0x9000:0x0080 ➔ es:di
        mov     cx,#0x10      ! 共传输 0x10 字节。
        rep
        movsb

! Get hd1 data

        mov     ax,#0x0000
        mov     ds,ax
        lds     si,[4*0x46]   ! 取中断向量 0x46 的值，也即 hd1 参数表的地址➔ds:si
```

```
        mov     ax,#INITSEG
        mov     es,ax
        mov     di,#0x0090   ! 传输的目的地址: 0x9000:0x0090 → es:di
        mov     cx,#0x10
        rep
        movsb

! Check that there IS a hd1 :-)  ! 检查系统是否存在第 2 个硬盘，如果不存在则第 2 个表清零。
   ! 利用 BIOS 中断调用 0x13 的取盘类型功能。
   ! 功能号 ah = 0x15;
   ! 输入：dl = 驱动器号（0x8X 是硬盘：0x80 指第 1 个硬盘，0x81 第 2 个硬盘）
   ! 输出：ah = 类型码；00 --没有这个盘，CF 置位； 01 --是软驱，没有 change-line 支持；
   !       02 --是软驱(或其它可移动设备)，有 change-line 支持； 03 --是硬盘。

        mov     ax,#0x01500
        mov     dl,#0x81
        int     0x13
        jc      no_disk1
        cmp     ah,#3            ! 是硬盘吗？(类型 = 3 ？)。
        je      is_disk1
no_disk1:
        mov     ax,#INITSEG      ! 第 2 个硬盘不存在，则对第 2 个硬盘表清零。
        mov     es,ax
        mov     di,#0x0090
        mov     cx,#0x10
        mov     ax,#0x00
        rep
        stosb
is_disk1:

! now we want to move to protected mode ...   ! 从这里开始我们要保护模式方面的工作了。

        cli                     ! no interrupts allowed !  ! 此时不允许中断。

! first we move the system to it's rightful place
! 首先我们将 system 模块移到正确的位置。
! bootsect 引导程序是将 system 模块读入到从 0x10000（64k）开始的位置。由于当时假设
! system 模块最大长度不会超过 0x80000（512k），也即其末端不会超过内存地址 0x90000，
! 所以 bootsect 会将自己移动到 0x90000 开始的地方，并把 setup 加载到它的后面。
! 下面这段程序的用途是再把整个 system 模块移动到 0x00000 位置，即把从 0x10000 到 0x8ffff
! 的内存数据块(512k)，整块地向内存低端移动了 0x10000（64k）的位置。

        mov     ax,#0x0000
        cld                     ! 'direction'=0, movs moves forward
do_move:
        mov     es,ax           ! destination segment ! es:di→目的地址(初始为 0x0000:0x0)
        add     ax,#0x1000
        cmp     ax,#0x9000      ! 已经把从 0x8000 段开始的 64k 代码移动完？
        jz      end_move
        mov     ds,ax           ! source segment  ! ds:si→源地址(初始为 0x1000:0x0)
        sub     di,di
        sub     si,si
        mov     cx,#0x8000      ! 移动 0x8000 字（64k 字节）。
```

```
124         rep
125         movsw
126         jmp     do_move
127
128 ! then we load the segment descriptors
    ! 此后，我们加载段描述符。
    ! 从这里开始会遇到 32 位保护模式的操作，因此需要 Intel 32 位保护模式编程方面的知识了,
    ! 有关这方面的信息请查阅列表后的简单介绍或附录中的详细说明。这里仅作概要说明。
    !
    ! lidt 指令用于加载中断描述符表(idt)寄存器，它的操作数是 6 个字节，0-1 字节是描述符表的
    ! 长度值(字节)；2-5 字节是描述符表的 32 位线性基地址（首地址），其形式参见下面
    ! 219-220 行和 223-224 行的说明。中断描述符表中的每一个表项（8 字节）指出发生中断时
    ! 需要调用的代码的信息，与中断向量有些相似，但要包含更多的信息。
    !
    ! lgdt 指令用于加载全局描述符表(gdt)寄存器，其操作数格式与 lidt 指令的相同。全局描述符
    ! 表中的每个描述符项(8 字节)描述了保护模式下数据和代码段（块）的信息。其中包括段的
    ! 最大长度限制(16 位)、段的线性基址（32 位）、段的特权级、段是否在内存、读写许可以及
    ! 其它一些保护模式运行的标志。参见后面 205-216 行。
    !
129
130 end_move:
131         mov     ax,#SETUPSEG    ! right, forgot this at first. didn't work :-)
132         mov     ds,ax           ! ds 指向本程序(setup)段。
133         lidt    idt_48          ! load idt with 0,0
                                    ! 加载中断描述符表(idt)寄存器，idt_48 是 6 字节操作数的位置
                                    ! (见 218 行)。前 2 字节表示 idt 表的限长，后 4 字节表示 idt 表
                                    ! 所处的基地址。
134         lgdt    gdt_48          ! load gdt with whatever appropriate
                                    ! 加载全局描述符表(gdt)寄存器，gdt_48 是 6 字节操作数的位置
                                    ! (见 222 行)。
135
136 ! that was painless, now we enable A20
    ! 以上的操作很简单，现在我们开启 A20 地址线。参见程序列表后有关 A20 信号线的说明。
137
138         call    empty_8042              ! 等待输入缓冲器空。
                                            ! 只有当输入缓冲器为空时才可以对其进行写命令。
139         mov     al,#0xD1                ! command write ! 0xD1 命令码-表示要写数据到
140         out     #0x64,al                ! 8042 的 P2 端口。P2 端口的位 1 用于 A20 线的选通。
                                            ! 数据要写到 0x60 口。
141         call    empty_8042              ! 等待输入缓冲器空，看命令是否被接受。
142         mov     al,#0xDF                ! A20 on  ! 选通 A20 地址线的参数。
143         out     #0x60,al
144         call    empty_8042              ! 输入缓冲器为空，则表示 A20 线已经选通。
145
146 ! well, that went ok, I hope. Now we have to reprogram the interrupts :-(
147 ! we put them right after the intel-reserved hardware interrupts, at
148 ! int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
149 ! messed this up with the original PC, and they haven't been able to
150 ! rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
151 ! which is used for the internal hardware interrupts as well. We just
152 ! have to reprogram the 8259's, and it isn't fun.
    !! 希望以上一切正常。现在我们必须重新对中断进行编程☹
    !! 我们将它们放在正好处于 intel 保留的硬件中断后面，在 int 0x20-0x2F。
```

```
!! 在那里它们不会引起冲突。不幸的是 IBM 在原 PC 机中搞糟了，以后也没有纠正过来。
!! PC 机的 bios 将中断放在了 0x08-0x0f，这些中断也被用于内部硬件中断。
!! 所以我们就必须重新对 8259 中断控制器进行编程，这一点都没劲。

        mov     al,#0x11                ! initialization sequence
                                        ! 0x11 表示初始化命令开始，是 ICW1 命令字，表示边
                                        ! 沿触发、多片 8259 级连、最后要发送 ICW4 命令字。
        out     #0x20,al                ! send it to 8259A-1   ! 发送到 8259A 主芯片。
        .word   0x00eb,0x00eb           ! jmp $+2, jmp $+2     ! $ 表示当前指令的地址，
                                        ! 两条跳转指令，跳到下一条指令，起延时作用。
        out     #0xA0,al                ! and to 8259A-2       ! 再发送到 8259A 从芯片。
        .word   0x00eb,0x00eb
        mov     al,#0x20                ! start of hardware int's (0x20)
        out     #0x21,al                ! 送主芯片 ICW2 命令字，起始中断号，要送奇地址。
        .word   0x00eb,0x00eb
        mov     al,#0x28                ! start of hardware int's 2 (0x28)
        out     #0xA1,al                ! 送从芯片 ICW2 命令字，从芯片的起始中断号。
        .word   0x00eb,0x00eb
        mov     al,#0x04                ! 8259-1 is master
        out     #0x21,al                ! 送主芯片 ICW3 命令字，主芯片的 IR2 连从芯片 INT。
        .word   0x00eb,0x00eb           ! 参见代码列表后的说明。
        mov     al,#0x02                ! 8259-2 is slave
        out     #0xA1,al                ! 送从芯片 ICW3 命令字，表示从芯片的 INT 连到主芯
                                        ! 片的 IR2 引脚上。
        .word   0x00eb,0x00eb
        mov     al,#0x01                ! 8086 mode for both
        out     #0x21,al                ! 送主芯片 ICW4 命令字。8086 模式；普通 EOI 方式，
                                        ! 需发送指令来复位。初始化结束，芯片就绪。
        .word   0x00eb,0x00eb
        out     #0xA1,al                ! 送从芯片 ICW4 命令字，内容同上。
        .word   0x00eb,0x00eb
        mov     al,#0xFF                ! mask off all interrupts for now
        out     #0x21,al                ! 屏蔽主芯片所有中断请求。
        .word   0x00eb,0x00eb
        out     #0xA1,al                ! 屏蔽从芯片所有中断请求。

! well, that certainly wasn't fun :-(. Hopefully it works, and we don't
! need no steenking BIOS anyway (except for the initial loading :-).
! The BIOS-routine wants lots of unnecessary data, and it's less
! "interesting" anyway. This is how REAL programmers do it.
!
! Well, now's the time to actually move into protected mode. To make
! things as simple as possible, we do no register set-up or anything,
! we let the gnu-compiled 32-bit programs do that. We just jump to
! absolute address 0x00000, in 32-bit protected mode.
!! 哼，上面这段当然没劲☹，希望这样能工作，而且我们也不再需要乏味的 BIOS 了（除了
!! 初始的加载☺。BIOS 子程序要求很多不必要的数据，而且它一点都没趣。那是“真正”的
!! 程序员所做的事。

   ! 这里设置进入 32 位保护模式运行。首先加载机器状态字(lmsw - Load Machine Status Word)，也
称
   ! 控制寄存器 CR0，其比特位 0 置 1 将导致 CPU 工作在保护模式。
        mov     ax,#0x0001      ! protected mode (PE) bit ! 保护模式比特位(PE)。
```

```
192         lmsw    ax                  ! This is it! ! 就这样加载机器状态字!
193         jmpi    0,8                 ! jmp offset 0 of segment 8 (cs) ! 跳转至 cs 段 8，偏移 0 处。
    ! 我们已经将 system 模块移动到 0x00000 开始的地方，所以这里的偏移地址是 0。这里的段
    ! 值的 8 已经是保护模式下的段选择符了，用于选择描述符表和描述符表项以及所要求的
    ! 特权级。
    ! 段选择符长度为 16 位（2 字节）；位 0-1 表示请求的特权级 0-3，linux 操作系统只
    ! 用到两级：0 级（系统级）和 3 级（用户级）；位 2 用于选择全局描述符表(0)还是局部描
    ! 述符表(1)；位 3-15 是描述符表项的索引，指出选择第几项描述符。所以段选择符
    ! 8(0b0000,0000,0000,1000)表示请求特权级 0、使用全局描述符表中的第 1 项，该项指出
    ! 代码的基地址是 0（参见 209 行），因此这里的跳转指令就会去执行 system 中的代码。
194
195 ! This routine checks that the keyboard command queue is empty
196 ! No timeout is used - if this hangs there is something wrong with
197 ! the machine, and we probably couldn't proceed anyway.
    ! 下面这个子程序检查键盘命令队列是否为空。这里不使用超时方法 - 如果这里死机，
    ! 则说明 PC 机有问题，我们就没有办法再处理下去了。
    ! 只有当输入缓冲器为空时（状态寄存器位 2 = 0）才可以对其进行写命令。
198 empty_8042:
199         .word   0x00eb,0x00eb       ! 这是两个跳转指令的机器码(跳转到下一句)，相当于延时空操作。
200         in      al,#0x64            ! 8042 status port  ! 读 AT 键盘控制器状态寄存器。
201         test    al,#2               ! is input buffer full?  ! 测试位 2，输入缓冲器满?
202         jnz     empty_8042          ! yes - loop
203         ret
204
205 gdt:   ! 全局描述符表开始处。描述符表由多个 8 字节长的描述符项组成。
       ! 这里给出了 3 个描述符项。第 1 项无用（206 行），但须存在。第 2 项是系统代码段
       ! 描述符（208-211 行），第 3 项是系统数据段描述符(213-216 行)。每个描述符的具体
       ! 含义参见列表后说明。
206         .word   0,0,0,0             ! dummy   ! 第 1 个描述符，不用。
207   ! 这里在 gdt 表中的偏移量为 0x08，当加载代码段寄存器(段选择符)时，使用的是这个偏移值。
208         .word   0x07FF              ! 8Mb - limit=2047 (2048*4096=8Mb)
209         .word   0x0000              ! base address=0
210         .word   0x9A00              ! code read/exec
211         .word   0x00C0              ! granularity=4096, 386
212   ! 这里在 gdt 表中的偏移量是 0x10，当加载数据段寄存器(如 ds 等)时，使用的是这个偏移值。
213         .word   0x07FF              ! 8Mb - limit=2047 (2048*4096=8Mb)
214         .word   0x0000              ! base address=0
215         .word   0x9200              ! data read/write
216         .word   0x00C0              ! granularity=4096, 386
217
218 idt_48:
219         .word   0                   ! idt limit=0
220         .word   0,0                 ! idt base=0L
221
222 gdt_48:
223         .word   0x800               ! gdt limit=2048, 256 GDT entries
                                        ! 全局表长度为 2k 字节，因为每 8 字节组成一个段描述符项
                                        ! 所以表中共可有 256 项。
224         .word   512+gdt,0x9         ! gdt base = 0X9xxxx
                                        ! 4 个字节构成的内存线性地址：0x0009<<16 + 0x0200+gdt
                                        ! 也即 0x90200 + gdt(即在本程序段中的偏移地址，205 行)。
225
226 .text
```

```
endtext:
.data
enddata:
.bss
endbss:
```

3.3.3 其它信息

为了获取机器的基本参数,这段程序多次调用了 BIOS 中的中断,并开始涉及一些对硬件端口的操作。下面简要地描述了程序中使用到的 BIOS 中断调用，并对 A20 地址线问题的原由进行了解释，最后提及关于 Intel 32 位保护模式运行的问题。

3.3.3.1 BIOS 视频中断 0x10

这里说明上面程序中用到的 ROM BIOS 中视频中断调用的几个子功能。

A. 获取显示卡信息（其它辅助功能选择）:

表 3.2 **获取显示卡信息（功能号：** ah = 0x12，bh = 0x10）

输入/返回信息	寄存器	内容说明
输入信息	ah	功能号=0x12，获取显示卡信息
	bh	子功能号=0x10。
返回信息	bh	视频状态： 0x00 - 彩色模式（此时视频硬件 I/O 端口基地址为 0x3DX); 0x01 - 单色模式（此时视频硬件 I/O 端口基地址为 0x3BX); 注：其中端口地址中的 X 值可为 0 - f。
	bl	已安装的显示内存大小： 00 = 64K, 01 = 128K, 02 = 192K, 03 = 256K
	ch	特性连接器比特位信息： 比特位 说明 0 特性线 1，状态 2; 1 特性线 0，状态 2; 2 特性线 1，状态 1; 3 特性线 0，状态 1; 4-7 未使用(为 0)

	cl	视频开关设置信息： 比特位 说明 0 开关 1 关闭； 1 开关 2 关闭； 2 开关 3 关闭； 3 开关 4 关闭； 4-7 未使用。 原始 EGA/VGA 开关设置值: 0x00 MDA/HGC； 0x01-0x03 MDA/HGC； 0x04 CGA 40x25； 0x05 CGA 80x25； 0x06 EGA+ 40x25； 0x07-0x09 EGA+ 80x25； 0x0A EGA+ 80x25 单色； 0x0B EGA+ 80x25 单色。

3.3.3.2 硬盘基本参数表（“INT 0x41”）

中断向量表中，int 0x41 的中断向量位置（4 * 0x41 =0x0000:0x0104）存放的并不是中断程序的地址，而是第一个硬盘的基本参数表。对于 100%兼容的 BIOS 来说，这里存放着硬盘参数表阵列的首地址 F000h:E401h。第二个硬盘的基本参数表入口地址存于 int 0x46 中断向量位置处。

表 3.3 硬盘基本参数信息表

位移	大小	说明
0x00	字	柱面数
0x02	字节	磁头数
0x03	字	开始减小写电流的柱面(仅 PC XT 使用，其它为 0)
0x05	字	开始写前预补偿柱面号（乘 4）
0x07	字节	最大 ECC 猝发长度（仅 XT 使用，其它为 0）
0x08	字节	控制字节（驱动器步进选择） 位 0 未用 位 1 保留(0) (关闭 IRQ) 位 2 允许复位 位 3 若磁头数大于 8 则置 1 位 4 未用(0) 位 5 若在柱面数+1 处有生产商的坏区图，则置 1 位 6 禁止 ECC 重试 位 7 禁止访问重试。
0x09	字节	标准超时值（仅 XT 使用，其它为 0）
0x0A	字节	格式化超时值（仅 XT 使用，其它为 0）
0x0B	字节	检测驱动器超时值（仅 XT 使用，其它为 0）
0x0C	字	磁头着陆(停止)柱面号
0x0E	字节	每磁道扇区数
0x0F	字节	保留。

3.3.3.3 A20 地址线问题

1981 年 8 月，IBM 公司最初推出的个人计算机 IBM PC 使用的 CPU 是 Intel 8088。在该微机中地址线只有 20 根(A0 - A19)。在当时内存 RAM 只有几百 KB 或不到 1MB 时，20 根地址线已足够用来寻址这些内存。其所能寻址的最高地址是 0xffff:0xffff，也即 0x10ffef。对于超出 0x100000(1MB)的寻址地址将默认地环绕到 0x0ffef。当 IBM 公司于 1985 年引入 AT 机时，使用的是 Intel 80286 CPU，具有 24 根地址线，最高可寻址 16MB，并且有一个与 8088 完全兼容的实模式运行方式。然而，在寻址值超过 1MB 时它却不能象 8088 那样实现地址寻址的环绕。但是当时已经有一些程序是利用这种地址环绕机制进行工作的。为了实现完全的兼容性，IBM 公司发明了使用一个开关来开启或禁止 0x100000 地址比特位。由于在当时的 8042 键盘控制器上恰好有空闲的端口引脚（输出端口 P2，引脚 P21），于是便使用了该引脚来作为与门控制这个地址比特位。该信号即被称为 A20。如果它为零，则比特 20 及以上地址都被清除。从而实现了兼容性。

由于在机器启动时，默认条件下，A20 地址线是禁止的，所以操作系统必须使用适当的方法来开启它。但是由于各种兼容机所使用的芯片集不同，要做到这一点却是非常的麻烦。因此通常要在几种控制方法中选择。

对 A20 信号线进行控制的常用方法是通过设置键盘控制器的端口值。这里的 setup.s 程序（138-144 行）即使用了这种典型的控制方式。对于其它一些兼容微机还可以使用其它方式来做到对 A20 线的控制。

有些操作系统将 A20 的开启和禁止作为实模式与保护运行模式之间进行转换的标准过程中的一部分。由于键盘的控制器速度很慢，因此就不能使用键盘控制器对 A20 线来进行操作。为此引进了一个 A20 快速门选项(Fast Gate A20)，它使用 I/O 端口 0x92 来处理 A20 信号线，避免了使用慢速的键盘控制器操作方式。对于不含键盘控制器的系统就只能使用 0x92 端口来控制，但是该端口也有可能被其它兼容微机上的设备（如显示芯片）所使用，从而造成系统错误的操作。

还有一种方式是通过读 0xee 端口来开启 A20 信号线，写该端口则会禁止 A20 信号线。

3.3.3.4 8259 中断控制器芯片

8259A 是一种可编程的中断控制芯片，每片可以管理 8 个中断源。通过多片的级联方式，能构成最多管理 64 个中断向量的系统。在 PC/AT 系列兼容机中，使用了两片 8259A 芯片，共可管理 15 级中断向量。其级连示意图见图 3.2 所示。其中从芯片的 INT 引脚连接到主芯片的 IR2 上。主 8259A 芯片的端口基地址是 0x20，从芯片是 0xA0。

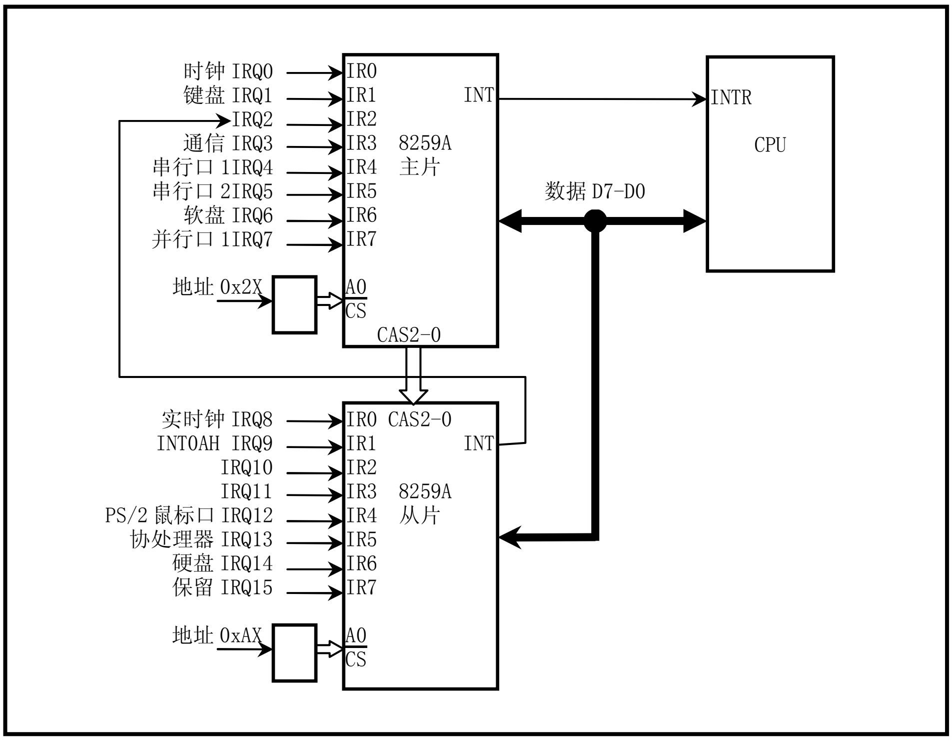


图 3.2 PC/AT 微机级连式 8259 控制系统

在总线控制器的控制下，芯片可以处于编程状态和操作状态。编程状态是 CPU 使用 IN 或 OUT 指令对 8259A 芯片进行初始化编程的状态。一旦完成了初始化编程，芯片即进入操作状态，此时芯片即可随时响应外部设备提出的中断请求（IRQ0 – IRQ15）。通过中断判优选择，芯片将选中当前最高优先级的中断请求作为中断服务对象，并通过 CPU 引脚 INT 通知 CPU 外中断请求的到来，CPU 响应后，芯片从数据总线 D7-D0 将编程设定的当前服务对象的中断号送出，CPU 由此获取对应的中断向量值，并执行中断服务程序。

3.3.3.5 Intel CPU 32 位保护运行模式

Intel CPU 一般可以在两种模式下运行，即实地址模式和保护模式。早期的 Intel CPU（8088/8086）只能工作在实模式下，某一时刻只能运行单个任务。对于 Intel 80386 以上的芯片则还可以运行在 32 位保护模式下。在保护模式下运行可以支持多任务；支持 4G 的物理内存；支持虚拟内存；支持内存的页式管理和段式管理；支持特权级。

虽然对保护模式下的运行机制是理解 Linux 内核的重要基础，但由于篇幅所限，对其工作原理的简单介绍可以参考书后的附录。但仍然建议初学者能够使用书后列出相关书籍，作一番仔细研究。为了真正理解 setup.s 程序和下面 head.s 程序的作用，起码要先明白段选择符、段描述符和 80x86 的页表寻址机制。

3.4 head.s 文件

3.4.1 功能描述

head.s 程序在被编译后，会被连接成 system 模块的最前面开始部分，这也就是为什么称其为头部(head)程序的原因。从这里开始，内核完全都是在保护模式下运行了。heads.s 汇编程序与前面的语法

格式不同，它采用的是 AT&T 的汇编语言格式，并且需要使用 GNU 的 gas 和 gld[1]进行编译连接。因此请注意代码中赋值的方向是从左到右。

这段程序实际上处于内存绝对地址 0 处开始的地方。这个程序的功能比较单一。首先是加载各个数据段寄存器，重新设置中断描述符表，共 256 项，并使各个表项均指向一个只报错误的哑中断程序。然后重新设置全局描述符表。接着使用物理地址 0 与 1M 开始处的内容相比较的方法，检测 A20 地址线是否已真的开启（如果没有开启，则在访问高于 1Mb 物理内存地址时 CPU 实际只会访问（IP MOD 1Mb）地址处的内容），如果检测下来发现没有开启，则进入死循环。然后程序测试 PC 机是否含有数学协处理器芯片（80287、80387 或其兼容芯片），并在控制寄存器 CR0 中设置相应的标志位。接着设置管理内存的分页处理机制，将页目录表放在绝对物理地址 0 开始处（也是本程序所处的物理内存位置，因此这段程序将被覆盖掉），紧随后面放置共可寻址 16MB 内存的 4 个页表，并分别设置它们的表项。最后利用返回指令将预先放置在堆栈中的/init/main.c 程序的入口地址弹出，去运行 main()程序。

3.4.2 代码注释

列表 3.4 linux/boot/head.s 程序

```
1 /*
2  *  linux/boot/head.s
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7 /*
8  *  head.s contains the 32-bit startup code.
9  *
10 * NOTE!!! Startup happens at absolute address 0x00000000, which is also where
11 * the page directory will exist. The startup code will be overwritten by
12 * the page directory.
13 */
   /*
    *  head.s 含有 32 位启动代码。
    * 注意!!! 32 位启动代码是从绝对地址 0x00000000 开始的，这里也同样是页目录将存在的地方，
   * 因此这里的启动代码将被页目录覆盖掉。
   */
14 .text
15 .globl _idt,_gdt,_pg_dir,_tmp_floppy_area
16 _pg_dir:   # 页目录将会存放在这里。
17 startup_32:                # 18-22 行设置各个数据段寄存器。
18         movl $0x10,%eax    # 对于 GNU 汇编来说，每个直接数要以'$'开始，否则是表示地址。
                              # 每个寄存器名都要以'%'开头，eax 表示是 32 位的 ax 寄存器。

   # 再次注意!!! 这里已经处于 32 位运行模式，因此这里的$0x10 并不是把地址 0x10 装入各个
   # 段寄存器，它现在其实是全局段描述符表中的偏移值，或者更正确地说是一个描述符表项
   # 的选择符。有关选择符的说明请参见 setup.s 中 193 行下的说明。这里$0x10 的含义是请求
   # 特权级 0(位 0-1=0)、选择全局描述符表(位 2=0)、选择表中第 1 项(位 3-15=1)。它正好指
   # 向表中的数据段描述符项。（描述符的具体数值参见前面 setup.s 中 212，213 行）
19         mov %ax,%ds
20         mov %ax,%es
```

[1] 在当前的 Linux 操作系统中，gas 和 gld 已经分别更名为 as 和 ld。

```
21          mov %ax,%fs
22          mov %ax,%gs
23          lss _stack_start,%esp     # 表示_stack_start➔ss:esp。_stack_start 应是编译程序
                                      # 为程序自动生成的存有堆栈信息地方。
24          call setup_idt            # 调用设置中断描述符表子程序。
25          call setup_gdt            # 调用设置全局描述符表子程序。
26          movl $0x10,%eax           # reload all the segment registers
                                      # 重新加载所有的段寄存器。
                                      # 在改变过全局描述符表 gdt 之后，CS 代码段寄存器已经
                                      # 在 setup_gdt 中重新加载过了。
27          mov %ax,%ds               # after changing gdt. CS was already
28          mov %ax,%es               # reloaded in 'setup_gdt'
29          mov %ax,%fs
30          mov %ax,%gs
31          lss _stack_start,%esp
   # 32-36 行用于测试 A20 地址线是否已经开启。采用的方法是向内存地址 0x000000 处写入任意
   # 一个数值，然后看内存地址 0x100000(1M)处是否也是这个数值。如果一直相同的话，就一直
   # 比较下去，也即死循环、死机。表示地址 A20 线没有选通，结果不能对 1M 以上内存寻址。
32          xorl %eax,%eax
33 1:       incl %eax                 # check that A20 really IS enabled
34          movl %eax,0x000000        # loop forever if it isn't
35          cmpl %eax,0x100000
36          je 1b                     # '1b'表示向后(backward)跳转到标号 1 去（33 行）。
                                      # 若是'5f'则表示向前(forward)跳转到标号 5 去。
37 /*
38  * NOTE! 486 should set bit 16, to check for write-protect in supervisor
39  * mode. Then it would be unnecessary with the "verify_area()"-calls.
40  * 486 users probably want to set the NE (#5) bit also, so as to use
41  * int 16 for math errors.
42  */
   /*
    * 注意！ 在下面这段程序中，486 应该将位 16 置位，以检查在超级用户模式下的写保护,
    * 此后"verify_area()"调用中就不需要了。486 的用户通常也会想将 NE(#5)置位，以便
    * 对数学协处理器的出错使用 int 16。
    */
    # 下面这段程序（43-65）用于检查数学协处理器芯片是否存在。方法是修改控制寄存器 CR0，在
    # 假设存在协处理器的情况下执行一个协处理器指令，如果出错的话则说明协处理器芯片不存在，
   # 需要设置 CR0 中的协处理器仿真位 EM（位 2），并复位协处理器存在标志 MP（位 1）。
43          movl %cr0,%eax            # check math chip
44          andl $0x80000011,%eax     # Save PG,PE,ET
45 /* "orl $0x10020,%eax" here for 486 might be good */
46          orl $2,%eax               # set MP
47          movl %eax,%cr0
48          call check_x87
49          jmp after_page_tables     # 跳转到 135 行。
50
51 /*
52  * We depend on ET to be correct. This checks for 287/387.
53  */
   /*
    * 我们依赖于 ET 标志的正确性来检测 287/387 存在与否。
    */
54 check_x87:
```

```
        fninit
        fstsw %ax
        cmpb $0,%al
        je 1f                   /* no coprocessor: have to set bits */
        movl %cr0,%eax           # 如果存在的则向前跳转到标号 1 处，否则改写 cr0。
        xorl $6,%eax            /* reset MP, set EM */
        movl %eax,%cr0
        ret
.align 2    # 这里".align 2"的含义是指存储边界对齐调整。"2"表示调整到地址最后 2 位为零，
            # 即按 4 字节方式对齐内存地址。
1:      .byte 0xDB,0xE4         /* fsetpm for 287, ignored by 387 */  # 287 协处理器码。
        ret

/*
 *  setup_idt
 *
 *  sets up a idt with 256 entries pointing to
 *  ignore_int, interrupt gates. It then loads
 *  idt. Everything that wants to install itself
 *  in the idt-table may do so themselves. Interrupts
 *  are enabled elsewhere, when we can be relatively
 *  sure everything is ok. This routine will be over-
 *  written by the page tables.
 */
/*
 * 下面这段是设置中断描述符表子程序 setup_idt
 *
 * 将中断描述符表 idt 设置成具有 256 个项，并都指向 ignore_int 中断门。然后加载中断
 * 描述符表寄存器(用 lidt 指令)。真正实用的中断门以后再安装。当我们在其它地方认为一切
 * 都正常时再开启中断。该子程序将会被页表覆盖掉。
 */
 # 中断描述符表中的项虽然也是 8 字节组成，但其格式与全局表中的不同，被称为门描述符
 # (Gate Descriptor)。它的 0-1,6-7 字节是偏移量，2-3 字节是选择符，4-5 字节是一些标志。
setup_idt:
        lea ignore_int,%edx      # 将 ignore_int 的有效地址（偏移值）值➔edx 寄存器
        movl $0x00080000,%eax    # 将选择符 0x0008 置入 eax 的高 16 位中。
        movw %dx,%ax            /* selector = 0x0008 = cs */
                                 # 偏移值的低 16 位置入 eax 的低 16 位中。此时 eax 含有
                                 #门描述符低 4 字节的值。
        movw $0x8E00,%dx        /* interrupt gate - dpl=0, present */
                                 # 此时 edx 含有门描述符高 4 字节的值。
        lea _idt,%edi            # _idt 是中断描述符表的地址。
        mov $256,%ecx
rp_sidt:
        movl %eax,(%edi)         # 将哑中断门描述符存入表中。
        movl %edx,4(%edi)
        addl $8,%edi             # edi 指向表中下一项。
        dec %ecx
        jne rp_sidt
        lidt idt_descr           # 加载中断描述符表寄存器值。
        ret

/*
```

```
96  *  setup_gdt
97  *
98  *  This routines sets up a new gdt and loads it.
99  *  Only two entries are currently built, the same
100 *  ones that were built in init.s. The routine
101 *  is VERY complicated at two whole lines, so this
102 *  rather long comment is certainly needed :-).
103 *  This routine will beoverwritten by the page tables.
104 */
    /*
     * 设置全局描述符表项 setup_gdt
     * 这个子程序设置一个新的全局描述符表 gdt，并加载。此时仅创建了两个表项，与前
     * 面的一样。该子程序只有两行，“非常的”复杂，所以当然需要这么长的注释了☺。
105 setup_gdt:
106         lgdt gdt_descr      # 加载全局描述符表寄存器(内容已设置好，见 232-238 行)。
107         ret
108
109 /*
110  * I put the kernel page tables right after the page directory,
111  * using 4 of them to span 16 Mb of physical memory. People with
112  * more than 16MB will have to expand this.
113  */
    /* Linus 将内核的内存页表直接放在页目录之后，使用了 4 个表来寻址 16 Mb 的物理内存。
     * 如果你有多于 16 Mb 的内存，就需要在这里进行扩充修改。
     */
     # 每个页表长为 4 Kb 字节，而每个页表项需要 4 个字节，因此一个页表共可以存放 1000 个表项，
     # 如果一个表项寻址 4 Kb 的地址空间，则一个页表就可以寻址 4 Mb 的物理内存。
     # 页表项的格式为：项的前 0-11 位存放一些标志，如是否在内存中(P 位 0)、读写许可(R/W 位 1)、
     # 普通用户还是超级用户使用(U/S 位 2)、是否修改过(是否脏了)(D 位 6)等；表项的位 12-31 是
     # 页框地址，用于指出一页内存的物理起始地址。
114 .org 0x1000      # 从偏移 0x1000 处开始是第 1 个页表（偏移 0 开始处将存放页表目录）。
115 pg0:
116
117 .org 0x2000
118 pg1:
119
120 .org 0x3000
121 pg2:
122
123 .org 0x4000
124 pg3:
125
126 .org 0x5000        # 定义下面的内存数据块从偏移 0x5000 处开始。
127 /*
128  * tmp_floppy_area is used by the floppy-driver when DMA cannot
129  * reach to a buffer-block. It needs to be aligned, so that it isn't
130  * on a 64kB border.
131  */
    /* 当 DMA（直接存储器访问）不能访问缓冲块时，下面的 tmp_floppy_area 内存块
     * 就可供软盘驱动程序使用。其地址需要对齐调整，这样就不会跨越 64kB 边界。
     */
132 _tmp_floppy_area:
133         .fill 1024,1,0          # 共保留 1024 项，每项 1 字节，填充数值 0。
```

```
134
      # 下面这几个入栈操作(pushl)用于为调用/init/main.c 程序和返回作准备。
      # 前面 3 个入栈指令不知道作什么用的，也许是 Linus 用于在调试时能看清机器码用的☺。
      # 139 行的入栈操作是模拟调用 main.c 程序时首先将返回地址入栈的操作，所以如果
      # main.c 程序真的退出时，就会返回到这里的标号 L6 处继续执行下去，也即死循环。
      # 140 行将 main.c 的地址压入堆栈，这样，在设置分页处理（setup_paging）结束后
      # 执行'ret'返回指令时就会将 main.c 程序的地址弹出堆栈，并去执行 main.c 程序去了。
135 after_page_tables:
136         pushl $0                # These are the parameters to main :-)
137         pushl $0                # 这些是调用 main 程序的参数（指 init/main.c）。
138         pushl $0
139         pushl $L6               # return address for main, if it decides to.
140         pushl $_main            # '_main'是编译程序对 main 的内部表示方法。
141         jmp setup_paging        # 跳转至第 198 行。
142 L6:
143         jmp L6                  # main should never return here, but
144                                 # just in case, we know what happens.
145
146 /* This is the default interrupt "handler" :-) */
    /* 下面是默认的中断“向量句柄”☺ */
147 int_msg:
148         .asciz "Unknown interrupt\n\r"     # 定义字符串“未知中断(回车换行)”。
149 .align 2                    # 按 4 字节方式对齐内存地址。
150 ignore_int:
151         pushl %eax
152         pushl %ecx
153         pushl %edx
154         push %ds            # 这里请注意！！ds,es,fs,gs 等虽然是 16 位的寄存器，但入栈后
                                # 仍然会以 32 位的形式入栈，也即需要占用 4 个字节的堆栈空间。
155         push %es
156         push %fs
157         movl $0x10,%eax     # 置段选择符（使 ds,es,fs 指向 gdt 表中的数据段）。
158         mov %ax,%ds
159         mov %ax,%es
160         mov %ax,%fs
161         pushl $int_msg      # 把调用 printk 函数的参数指针（地址）入栈。
162         call _printk        # 该函数在/kernel/printk.c 中。
                                # '_printk'是 printk 编译后模块中的内部表示法。
163         popl %eax
164         pop %fs
165         pop %es
166         pop %ds
167         popl %edx
168         popl %ecx
169         popl %eax
170         iret            # 中断返回（把中断调用时压入栈的 CPU 标志寄存器（32 位）值也弹出）。
171
172
173 /*
174  * Setup_paging
175  *
176  * This routine sets up paging by setting the page bit
177  * in cr0. The page tables are set up, identity-mapping
```

```
178  * the first 16MB. The pager assumes that no illegal
179  * addresses are produced (ie >4Mb on a 4Mb machine).
180  *
181  * NOTE! Although all physical memory should be identity
182  * mapped by this routine, only the kernel page functions
183  * use the >1Mb addresses directly. All "normal" functions
184  * use just the lower 1Mb, or the local data space, which
185  * will be mapped to some other place - mm keeps track of
186  * that.
187  *
188  * For those with more memory than 16 Mb - tough luck. I've
189  * not got it, why should you :-) The source is here. Change
190  * it. (Seriously - it shouldn't be too difficult. Mostly
191  * change some constants etc. I left it at 16Mb, as my machine
192  * even cannot be extended past that (ok, but it was cheap :-)
193  * I've tried to show which constants to change by having
194  * some kind of marker at them (search for "16Mb"), but I
195  * won't guarantee that's all :-( )
196  */
    /*
     * 这个子程序通过设置控制寄存器 cr0 的标志（PG 位 31）来启动对内存的分页处理功能，
     * 并设置各个页表项的内容，以恒等映射前 16 MB 的物理内存。分页器假定不会产生非法的
     * 地址映射（也即在只有 4Mb 的机器上设置出大于 4Mb 的内存地址）。
     * 注意！尽管所有的物理地址都应该由这个子程序进行恒等映射，但只有内核函数直接使用
     * >1Mb 的地址。所有“一般”函数仅使用低于 1Mb 的地址空间，或者是使用局部数据空间，
     * 地址空间将被映射到其它一些地方去 -- mm(内存管理程序)会管理这些事的。
     * 对于那些有多于 16Mb 内存的家伙 - 太幸运了，我还没有，为什么你会有☺。代码就在这里，
     * 对它进行修改吧。（实际上，这并不太困难的。通常只需修改一些常数等。我把它设置为
     * 16Mb，因为我的机器再怎么扩充甚至不能超过这个界限（当然，我的机器很便宜的☺）。
     * 我已经通过设置某类标志来给出需要改动的地方（搜索“16Mb”），但我不能保证作这些
     * 改动就行了☹）。
     */
197 .align 2          # 按 4 字节方式对齐内存地址边界。
198 setup_paging:     # 首先对 5 页内存（1 页目录 + 4 页页表）清零
199         movl $1024*5,%ecx               /* 5 pages - pg_dir+4 page tables */
200         xorl %eax,%eax
201         xorl %edi,%edi                  /* pg_dir is at 0x000 */
                                             # 页目录从 0x000 地址开始。
202         cld;rep;stosl
    # 下面 4 句设置页目录中的项，我们共有 4 个页表所以只需设置 4 项。
    # 页目录项的结构与页表中项的结构一样，4 个字节为 1 项。参见上面 113 行下的说明。
    # "$pg0+7"表示：0x00001007，是页目录表中的第 1 项。
    # 则第 1 个页表所在的地址 = 0x00001007 & 0xfffff000 = 0x1000;
    # 第 1 个页表的属性标志 = 0x00001007 & 0x00000fff = 0x07，表示该页存在、用户可读写。
203         movl $pg0+7,_pg_dir             /* set present bit/user r/w */
204         movl $pg1+7,_pg_dir+4           /*  --------- " " --------- */
205         movl $pg2+7,_pg_dir+8           /*  --------- " " --------- */
206         movl $pg3+7,_pg_dir+12          /*  --------- " " --------- */
    # 下面 6 行填写 4 个页表中所有项的内容，共有：4(页表)*1024(项/页表)=4096 项(0 - 0xfff)，
    # 也即能映射物理内存 4096*4Kb = 16Mb。
    # 每项的内容是：当前项所映射的物理内存地址 + 该页的标志（这里均为 7）。
    # 使用的方法是从最后一个页表的最后一项开始按倒退顺序填写。一个页表的最后一项在页表中的
    # 位置是 1023*4 = 4092。因此最后一页的最后一项的位置就是$pg3+4092。
```

```
        movl $pg3+4092,%edi              # edi➔最后一页的最后一项。
        movl $0xfff007,%eax             /*  16Mb - 4096 + 7 (r/w user,p) */
                                         # 最后 1 项对应物理内存页面的地址是 0xfff000，
                                         # 加上属性标志 7，即为 0xfff007.
        std                     # 方向位置位，edi 值递减(4 字节)。
1:      stosl                   /* fill pages backwards - more efficient :-) */
        subl $0x1000,%eax       # 每填写好一项，物理地址值减 0x1000。
        jge 1b                  # 如果小于 0 则说明全添写好了。
# 设置页目录表基址寄存器 cr3 的值，指向页目录表。
        xorl %eax,%eax          /* pg_dir is at 0x0000 */    # 页目录表在 0x0000 处。
        movl %eax,%cr3          /* cr3 - page directory start */
# 设置启动使用分页处理（cr0 的 PG 标志，位 31）
        movl %cr0,%eax
        orl $0x80000000,%eax    # 添上 PG 标志。
        movl %eax,%cr0          /* set paging (PG) bit */
        ret                     /* this also flushes prefetch-queue */
# 在改变分页处理标志后要求使用转移指令刷新预取指令队列，这里用的是返回指令 ret。
# 该返回指令的另一个作用是将堆栈中的 main 程序的地址弹出，并开始运行/init/main.c 程序。
# 本程序到此真正结束了。

.align 2                        # 按 4 字节方式对齐内存地址边界。
.word 0
idt_descr:                      #下面两行是 lidt 指令的 6 字节操作数：长度，基址。
        .word 256*8-1           # idt contains 256 entries
        .long _idt
.align 2
.word 0
gdt_descr:                      # 下面两行是 lgdt 指令的 6 字节操作数：长度，基址。
        .word 256*8-1           # so does gdt (not that that's any
        .long _gdt              # magic number, but it works for me :^)

        .align 3                # 按 8 字节方式对齐内存地址边界。
_idt:   .fill 256,8,0           # idt is uninitialized    # 256 项，每项 8 字节，填 0。

# 全局表。前 4 项分别是空项（不用）、代码段描述符、数据段描述符、系统段描述符，其中
# 系统段描述符 linux 没有派用处。后面还预留了 252 项的空间，用于放置所创建任务的
# 局部描述符(LDT)和对应的任务状态段 TSS 的描述符。
# (0-nul, 1-cs, 2-ds, 3-sys, 4-TSS0, 5-LDT0, 6-TSS1, 7-LDT1, 8-TSS2 etc...)
_gdt:   .quad 0x0000000000000000        /* NULL descriptor */
        .quad 0x00c09a0000000fff        /* 16Mb */  # 代码段最大长度 16M。
        .quad 0x00c0920000000fff        /* 16Mb */  # 数据段最大长度 16M。
        .quad 0x0000000000000000        /* TEMPORARY - don't use */
        .fill 252,8,0                   /* space for LDT's and TSS's etc */
```

3.4.3 其它信息

理解这段程序的关键是真正了解 Intel 386 32 位保护模式的运行机制，也是继续阅读以下其余程序所必须的。为了与 8086 CPU 兼容，80x86 的保护模式被处理的较为复杂。当 CPU 运行在保护模式下时，它就将实模式下的段地址当作保护模式下段描述符的指针使用，此时段寄存器中存放的是一个描述符在描述符表中的偏移地址值。而当前描述符表的基地址则保存在描述符表寄存器中，如全局描述符表寄存器 gdtr、中断门描述符表寄存器 idtr，加载这些表寄存器须使用专用指令 lgdt 或 lidt。

CPU 在实模式运行方式时，段寄存器用来放置一个内存段地址（比如 0x9000），而此时在该段内可以

寻址 64KB 的内存。但当进入保护模式运行方式时，此时段寄存器中放置的并不是内存中的某个地址值，而是指定描述符表中某个描述符项相对于该描述符表基址的一个偏移量。在这个 8 字节的描述符中含有该段线性地址的‘段’基址和段的长度，以及其它一些描述该段特征的比特位。因此此时所寻址的内存位置是这个段基址加上当前执行代码指针 eip 的值。当然，此时所寻址的实际物理内存地址，还需要经过内存页面处理管理机制进行变换后才能得到。简而言之，32 位保护模式下的内存寻址需要拐个弯，经过描述符表中的描述符和内存页管理来确定。

针对不同的使用方面，描述符表分为三种：全局描述符表（GDT）、中断描述符表（IDT）和局部描述符表（LDT）。当 CPU 运行在保护模式下，某一时刻 GDT 和 IDT 分别只能有一个，分别由寄存器 GDTR 和 IDTR 指定它们的表基址。局部表可以有 0-8191 个，其基址由当前 LDTR 寄存器的内容指定，是使用 GDT 中某个描述符来加载的，也即 LDT 也是由 GDT 中的描述符来指定。但是在某一时刻同样也只有其中的一个被认为是活动的。一般对于每个任务（进程）使用一个 LDT。在运行时，程序可以使用 GDT 中的描述符以及当前任务的 LDT 中的描述符。

中断描述符表 IDT 的结构与 GDT 类似，在 Linux 内核中它正好位于 GDT 表的后面。共含有 256 项 8 字节的描述符。但每个描述符项的格式与 GDT 的不同，其中存放着相应中断过程的偏移值（0-1，6-7 字节）、所处段的选择符值（2-3 字节）和一些标志（4-5 字节）。

下图 3.3 是 Linux 内核中所使用的描述符表在内存中的示意图。图中，每个任务在 GDT 中占有两个描述符项。GDT 表中的 LDT0 描述符项是第一个任务（进程）的局部描述符表的描述符，TSS0 是第一个任务的任务状态段（TSS）的描述符。每个 LDT 中含有三个描述符，其中第一个不用，第二个是任务代码段的描述符，第三个是任务数据段和堆栈段的描述符。当 DS 段寄存器中是第一个任务的数据段选择符时，DS:ESI 即指向该任务数据段中的某个数据。

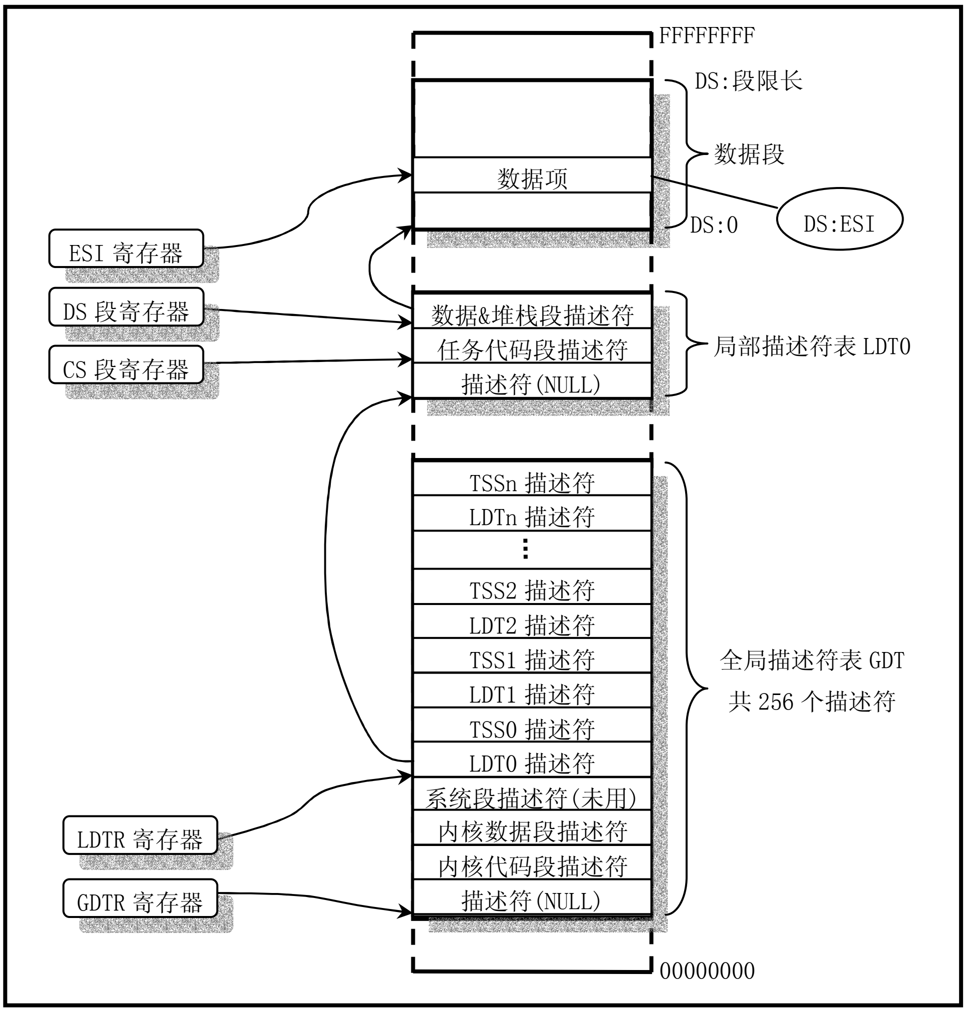


图 3.3 Linux 内核使用描述符表的示意图。

3.5 本章小结

在引导加载程序 bootsect.s 主要将 setup.s 代码和 system 模块加载到内存中，其中 system 模块的首部包含有 head.s 代码。在把自己移动到物理地址 0x90000 处并将 setup.s 代码放到 0x90200 处后，就将执行权交给了 setup 程序。

setup 程序的主要作用是利用 ROM BIOS 的中断程序获取机器的一些基本参数，并保存在 0x90000 开始的内存块中，供后面程序使用。同时把 system 模块往下移动到物理地址 0x00000 开始处，这样，system 中的 head.s 代码就处在 0x00000 开始处了。然后加载描述符表基地址到描述符表寄存器中，为进行 32 位保护模式下的运行作好准备。接下来对中断控制硬件进行重新设置，最后通过设置机器控制寄存器 CR0 并跳转到 system 模块的 head.s 代码开始处，使 CPU 进入 32 位保护模式下运行。

Head.s 代码的主要作用是初步初始化中断描述符表中的 256 项门描述符，检查 A20 地址线是否已经打开，测试系统是否含有数学协处理器。然后初始化内存页目录表，为内存的分页管理作好准备工作。最后跳转到 system 模块中的初始化程序 init.c 中继续执行。

下一章的主要内容就是详细描述 init.c 程序的功能和作用。

第 4 章 初始化程序(init)

4.1 概述

在内核源代码的 init/目录中只有一个 main.c 文件。系统在执行完 boot/目录中的 head.s 程序后就会将执行权交给 main.c。该程序虽然不长，但却包括了内核初始化的所有工作。因此在阅读该程序的代码时需要参照很多其它程序中的初始化部分。如果能完全理解这里调用的所有程序，那么看完这章内容后你应该对 Linux 内核有了大致的了解。

从这一章开始，我们将接触大量的 C 程序代码，因此读者最好具有一定的 C 语言知识。最好的一本参考书还是 Brian W. Kernighan 和 Dennis M. Ritchie 编著的《C 程序设计语言》，对该书第五章关于指针和数组的理解，可以说是弄懂 C 语言的关键。

在注释 C 语言程序时，为了与程序中原有的注释相区别，我们使用'//'作为注释语句的开始。有关原有注释的翻译则采用与其一样的注释标志。对于程序中包含的头文件（*.h），仅作概要含义的解释，具体详细注释内容将在注释相应头文件的章节中给出。

4.2 main.c 程序

4.2.1 功能描述

首先利用 setup.s 程序取得的系统参数设置系统的根文件设备号以及一些内存全局变量。这些内存变量指明了主内存的开始地址、系统所拥有的内存容量和作为高速缓冲区内存的末端地址。如果还定义了虚拟盘（RAMDISK），则主内存将适当减少。整个内存的印象示意图见图 4.1 所示。

内核程序	高速缓冲	虚拟盘	主内存区

图 4.1 系统中内存功能划分示意图。

然后，内核进行所有方面的硬件初始化工作。包括陷阱门、块设备、字符设备和 tty，包括人工创建第一个任务（task 0）。待所有初始化工作完成就设置中断允许标志，开启中断。在阅读这些初始化子程序时，最好是跟着被调用的程序深入进去看，如果实在看不下去了，就暂时先放一放，继续看下一个初始化调用。在有些理解之后再继续研究没有看完的地方。

在整个内核完成初始化后，内核将执行权切换到了用户模式，也即 CPU 从 0 特权级切换到了第 3 特权级。然后系统第一次调用创建进程函数 fork()，创建出一个用于运行 init()的子进程。

在该进程（任务）中系统将运行控制台程序。如果控制台环境建立成功，则再生成一个子进程，用于运行 shell 程序/bin/sh。若该子进程退出，父进程返回，则父进程进入一个死循环内，继续生成子进程，并在此子进程中再次执行 shell 程序/bin/sh，而父进程则继续等待。

对于 Linux 来说，所有任务都是在用户模式执行的，包括很多系统应用程序，如 shell 程序、网络

子系统程序等。

4.2.2 代码注释

列表 4.1 linux/init/main.c 程序

```
/*
 *  linux/init/main.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define __LIBRARY__        // 定义该变量是为了包括定义在 unistd.h 中的内嵌汇编代码等信息。
#include <unistd.h>        // *.h 头文件所在的默认目录是 include/，则在代码中就不用明确指明位置。
                           // 如果不是 UNIX 的标准头文件，则需要指明所在的目录，并用双引号括住。
                           // 标准符号常数与类型文件。定义了各种符号常数和类型，并申明了各种函数。
                           // 如果定义了__LIBRARY__，则还包括系统调用号和内嵌汇编代码_syscall0()
等。
#include <time.h>          // 时间类型头文件。其中最主要定义了 tm 结构和一些有关时间的函数原形。

/*
 * we need this inline - forking from kernel space will result
 * in NO COPY ON WRITE (!!!), until an execve is executed. This
 * is no problem, but for the stack. This is handled by not letting
 * main() use the stack at all after fork(). Thus, no function
 * calls - which means inline code for fork too, as otherwise we
 * would use the stack upon exit from 'fork()'.
 *
 * Actually only pause and fork are needed inline, so that there
 * won't be any messing with the stack from main(), but we define
 * some others too.
 */
/*
 * 我们需要下面这些内嵌语句 - 从内核空间创建进程(forking)将导致没有写时复制（COPY ON
WRITE）!!!
 * 直到一个执行 execve 调用。这对堆栈可能带来问题。处理的方法是在 fork()调用之后不让 main()使
用
 * 任何堆栈。因此就不能有函数调用 - 这意味着 fork 也要使用内嵌的代码，否则我们在从 fork()退出
 * 时就要使用堆栈了。
 * 实际上只有 pause 和 fork 需要使用内嵌方式，以保证从 main()中不会弄乱堆栈，但是我们同时还
 * 定义了其它一些函数。
 */
static inline _syscall0(int,fork)      // 是 unistd.h 中的内嵌宏代码。以嵌入汇编的形式调用
                                       // Linux 的系统调用中断 0x80。该中断是所有系统调用的
                                       // 入口。该条语句实际上是 int fork()创建进程系统调用。
                                       // syscall0 名称中最后的 0 表示无参数，1 表示 1 个参数。
static inline _syscall0(int,pause)           // int pause()系统调用：暂停进程的执行，直到
                                             // 收到一个信号。
static inline _syscall1(int,setup,void *,BIOS)   // int setup(void * BIOS)系统调用，仅用于
                                                 // linux 初始化（仅在这个程序中被调用）。
static inline _syscall0(int,sync)            // int sync()系统调用：更新文件系统。

#include <linux/tty.h>     // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
```

```
29 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、第 1 个初始任务
                             // 的数据。还有一些以宏的形式定义的有关描述符参数设置和获取的
                             // 嵌入式汇编函数程序。
30 #include <linux/head.h>   // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
31 #include <asm/system.h>   // 系统头文件。以宏的形式定义了许多有关设置或修改
                             // 描述符/中断门等的嵌入式汇编子程序。
32 #include <asm/io.h>       // io 头文件。以宏的嵌入汇编程序形式定义对 io 端口操作的函数。
33
34 #include <stddef.h>       // 标准定义头文件。定义了 NULL, offsetof(TYPE, MEMBER)。
35 #include <stdarg.h>       // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了-个
                             // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，vsprintf、
                             // vprintf、vfprintf。
36 #include <unistd.h>
37 #include <fcntl.h>        // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
38 #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
39
40 #include <linux/fs.h>     // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
41
42 static char printbuf[1024];  // 静态字符串数组。
43
44 extern int vsprintf();     // 送格式化输出到一字符串中（在 kernel/vsprintf.c，92 行）。
45 extern void init(void);    // 函数原形，初始化（在 168 行）。
46 extern void blk_dev_init(void);  // 块设备初始化子程序（kernel/blk_drv/ll_rw_blk.c,157 行）
47 extern void chr_dev_init(void);  // 字符设备初始化（kernel/chr_drv/tty_io.c, 347 行）
48 extern void hd_init(void);       // 硬盘初始化程序（kernel/blk_drv/hd.c, 343 行）
49 extern void floppy_init(void);   // 软驱初始化程序（kernel/blk_drv/floppy.c, 457 行）
50 extern void mem_init(long start, long end);  // 内存管理初始化（mm/memory.c, 399 行）
51 extern long rd_init(long mem_start, int length); //虚拟盘初始化(kernel/blk_drv/ramdisk.c,52)
52 extern long kernel_mktime(struct tm * tm);  // 建立内核时间（秒）。
53 extern long startup_time;  // 内核启动时间（开机时间）（秒）。
54
55 /*
56  * This is set up by the setup-routine at boot-time
57  */
   /*
    * 以下这些数据是由 setup.s 程序在引导时间设置的（参见第 2 章 2.3.1 节中的表 2.1）。
    */
58 #define EXT_MEM_K (*(unsigned short *)0x90002)        // 1M 以后的扩展内存大小（KB）。
59 #define DRIVE_INFO (*(struct drive_info *)0x90080)    // 硬盘参数表基址。
60 #define ORIG_ROOT_DEV (*(unsigned short *)0x901FC)    // 根文件系统所在设备号。
61
62 /*
63  * Yeah, yeah, it's ugly, but I cannot find how to do this correctly
64  * and this seems to work. I anybody has more info on the real-time
65  * clock I'd be interested. Most of this was trial and error, and some
66  * bios-listing reading. Urghh.
67  */
   /*
    * 是啊，是啊，下面这段程序很差劲，但我不知道如何正确地实现，而且好象它还能运行。如果有
    * 关于实时时钟更多的资料，那我很感兴趣。这些都是试探出来的，以及看了一些 bios 程序，呵！
    */
68
69 #define CMOS_READ(addr) ({ \    // 这段宏读取 CMOS 实时时钟信息。
```

```
outb_p(0x80|addr,0x70); \          // 0x70 是写端口号，0x80|addr 是要读取的 CMOS 内存地址。
inb_p(0x71); \                     // 0x71 是读端口号。
})

#define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10)   // 将 BCD 码转换成数字。

static void time_init(void)   // 该子程序取 CMOS 时钟，并设置开机时间→startup_time(秒)。
{
        struct tm time;

        do {
                time.tm_sec = CMOS_READ(0);       // 参见后面 CMOS 内存列表。
                time.tm_min = CMOS_READ(2);
                time.tm_hour = CMOS_READ(4);
                time.tm_mday = CMOS_READ(7);
                time.tm_mon = CMOS_READ(8);
                time.tm_year = CMOS_READ(9);
        } while (time.tm_sec != CMOS_READ(0));
        BCD_TO_BIN(time.tm_sec);
        BCD_TO_BIN(time.tm_min);
        BCD_TO_BIN(time.tm_hour);
        BCD_TO_BIN(time.tm_mday);
        BCD_TO_BIN(time.tm_mon);
        BCD_TO_BIN(time.tm_year);
        time.tm_mon--;
        startup_time = kernel_mktime(&time);
}

static long memory_end = 0;          // 机器具有的内存（字节数）。
static long buffer_memory_end = 0;   // 高速缓冲区末端地址。
static long main_memory_start = 0;   // 主内存（将用于分页）开始的位置。

struct drive_info { char dummy[32]; } drive_info;   // 用于存放硬盘参数表信息。

void main(void)           /* This really IS void, no error here. */
{                         /* The startup routine assumes (well, ...) this */
                          /* 这里确实是 void，并没错。在 startup 程序(head.s)中就是这样假设的。
                          // 参见 head.s 程序第 136 行开始的几行代码。
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
/*
 * 此时中断仍被禁止着，做完必要的设置后就将其开启。
 */
 // 下面这段代码用于保存：
 // 根设备号  →ROOT_DEV；    高速缓存末端地址→buffer_memory_end；
 // 机器内存数→memory_end；主内存开始地址  →main_memory_start；
        ROOT_DEV = ORIG_ROOT_DEV;
        drive_info = DRIVE_INFO;
        memory_end = (1<<20) + (EXT_MEM_K<<10); // 内存大小=1Mb 字节+扩展内存(k)*1024 字节。
        memory_end &= 0xfffff000;               // 忽略不到 4Kb（1 页）的内存数。
        if (memory_end > 16*1024*1024)          // 如果内存超过 16Mb，则按 16Mb 计。
```

```
115                 memory_end = 16*1024*1024;
116         if (memory_end > 12*1024*1024)          // 如果内存>12Mb，则设置缓冲区末端=4Mb
117                 buffer_memory_end = 4*1024*1024;
118         else if (memory_end > 6*1024*1024)      // 否则如果内存>6Mb，则设置缓冲区末端=2Mb
119                 buffer_memory_end = 2*1024*1024;
120         else
121                 buffer_memory_end = 1*1024*1024;// 否则则设置缓冲区末端=1Mb
122         main_memory_start = buffer_memory_end;  // 主内存起始位置=缓冲区末端;
123 #ifdef RAMDISK    // 如果定义了虚拟盘，则主内存将减少。
124         main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
125 #endif
     // 以下是内核进行所有方面的初始化工作。阅读时最好跟着调用的程序深入进去看，实在看
     // 不下去了，就先放一放，看下一个初始化调用 -- 这是经验之谈☺。
126         mem_init(main_memory_start,memory_end);
127         trap_init();      // 陷阱门（硬件中断向量）初始化。（kernel/traps.c，181 行）
128         blk_dev_init();   // 块设备初始化。     （kernel/blk_dev/ll_rw_blk.c，157 行）
129         chr_dev_init();   // 字符设备初始化。   （kernel/chr_dev/tty_io.c，347 行）
130         tty_init();       // tty 初始化。       （kernel/chr_dev/tty_io.c，105 行）
131         time_init();      // 设置开机启动时间➔startup_time（见 76 行）。
132         sched_init();     // 调度程序初始化(加载了任务 0 的 tr, ldtr) （kernel/sched.c，385）
133         buffer_init(buffer_memory_end);// 内存管理初始化，建内存链表等。（fs/buffer.c，348）
134         hd_init();        // 硬盘初始化。       （kernel/blk_dev/hd.c，343 行）
135         floppy_init();    // 软驱初始化。       （kernel/blk_dev/floppy.c，457 行）
136         sti();            // 所有初始化工作都做完了，开启中断。
137         move_to_user_mode();  // 移到用户模式。 （include/asm/system.h，第 1 行）
138         if (!fork()) {          /* we count on this going ok */
139                 init();
140         }
141 /*
142  *   NOTE!!   For any other task 'pause()' would mean we have to get a
143  * signal to awaken, but task0 is the sole exception (see 'schedule()')
144  * as task 0 gets activated at every idle moment (when no other tasks
145  * can run). For task0 'pause()' just means we go check if some other
146  * task can run, and if not we return here.
147  */
    /* 注意!! 对于任何其它的任务，'pause()'将意味着我们必须等待收到一个信号才会返
    * 回就绪运行态，但任务 0（task0）是唯一的意外情况（参见'schedule()'），因为任务 0 在
    * 任何空闲时间里都会被激活（当没有其它任务在运行时），因此对于任务 0'pause()'仅意味着
    * 我们返回来查看是否有其它任务可以运行，如果没有的话我们就回到这里，一直循环执行'pause()'。
    */
148         for(;;) pause();
149 }
150
151 static int printf(const char *fmt, ...)
    // 产生格式化信息并输出到标准输出设备 stdout(1)，这里是指屏幕上显示。参数'*fmt'指定输出将
    // 采用的格式，参见各种标准 C 语言书籍。该子程序正好是 vsprintf 如何使用的一个例子。
    // 该程序使用 vsprintf()将格式化的字符串放入 printbuf 缓冲区，然后用 write()将缓冲区的内容
    // 输出到标准设备（1--stdout）。
152 {
153         va_list args;
154         int i;
155
156         va_start(args, fmt);
```

```
157         write(1,printbuf,i=vsprintf(printbuf, fmt, args));
158         va_end(args);
159         return i;
160 }
161
162 static char * argv_rc[] = { "/bin/sh", NULL };  // 调用执行程序时参数的字符串数组。
163 static char * envp_rc[] = { "HOME=/", NULL };   // 调用执行程序时的环境字符串数组。
164
165 static char * argv[] = { "-/bin/sh",NULL };     // 同上。
166 static char * envp[] = { "HOME=/usr/root", NULL };
167
168 void init(void)
169 {
170         int pid,i;
171
172         setup((void *) &drive_info);    // 读取硬盘参数包括分区表信息并建立虚拟盘和
                                        // 安装根文件系统设备。(kernel/blk_drv/hd.c,71)
173         (void) open("/dev/tty0",O_RDWR,0);  // 用读写访问方式打开设备“/dev/tty0”，
                                            // 这里对应终端控制台。
                                            // 返回的句柄号 0 -- stdin 标准输入设备。
174         (void) dup(0);      // 复制句柄，产生句柄 1 号 -- stdout 标准输出设备。
175         (void) dup(0);      // 复制句柄，产生句柄 2 号 -- stderr 标准出错输出设备。
176         printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS,
177                 NR_BUFFERS*BLOCK_SIZE);    // 打印缓冲区块数和总字节数，每块 1024 字节。
178         printf("Free mem: %d bytes\n\r",memory_end-main_memory_start);  //空闲内存字节数。
    // 下面 fork()用于创建一个子进程(子任务)。对于被创建的子进程，fork()将返回 0 值，
    // 对于原(父进程)将返回子进程的进程号。所以 180-184 句是子进程执行的内容。该子进程
    // 关闭了句柄 0(stdin)，以只读方式打开/etc/rc 文件，并执行/bin/sh 程序，所带参数和
    // 环境变量分别由 argv_rc 和 envp_rc 数组给出。参见后面的描述。
179         if (!(pid=fork())) {
180                 close(0);
181                 if (open("/etc/rc",O_RDONLY,0))
182                         _exit(1);     // 如果打开文件失败，则退出(/lib/_exit.c,10)。
183                 execve("/bin/sh",argv_rc,envp_rc);  // 装入/bin/sh 程序并执行。
184                 _exit(2);
185         }
    // 下面是父进程执行的语句。wait()是等待子进程停止或终止，其返回值应是子进程的进程号(pid)。
    // 这三句的作用是父进程等待子进程的结束。&i 是存放返回状态信息的位置。如果 wait()返回值不
    // 等于子进程号，则继续等待。
186         if (pid>0)
187                 while (pid != wait(&i))
188                         /* nothing */;
    // 如果执行到这里，说明刚创建的子进程的执行已停止或终止了。下面循环中首先再创建一个子进程，
    // 如果出错，则显示“初始化程序创建子进程失败”的信息并继续执行。对于所创建的子进程关闭所
有
    // 以前还遗留的句柄(stdin, stdout, stderr)，新创建一个会话并设置进程组号，然后重新打开
    ///dev/tty0 作为 stdin，并复制成 stdout 和 stderr。再次执行系统解释程序/bin/sh。但这次执行所
    // 选用的参数和环境数组另选了一套（见上面 165-167 行）。然后父进程再次运行 wait()等待。如果
    // 子进程又停止了执行，则在标准输出上显示出错信息“子进程 pid 停止了运行，返回码是 i”，然后
    // 继续重试下去…，形成“大”死循环。
189         while (1) {
190                 if ((pid=fork())<0) {
191                         printf("Fork failed in init\r\n");
```

```
                        continue;
                }
                if (!pid) {
                        close(0);close(1);close(2);
                        setsid();
                        (void) open("/dev/tty0",O_RDWR,0);
                        (void) dup(0);
                        (void) dup(0);
                        _exit(execve("/bin/sh",argv,envp));
                }
                while (1)
                        if (pid == wait(&i))
                                break;
                printf("\n\rchild %d died with code %04x\n\r",pid,i);
                sync();
        }
        _exit(0);       /* NOTE! _exit, not exit() */
}

```

4.2.3 其它信息

4.2.3.1 CMOS 信息

PC 机的 CMOS(complementary metal oxide semiconductor 互补金属氧化物半导体)内存实际上是由电池供电的 64 或 128 字节 RAM 内存块，是系统时钟芯片的一部分。有些机器还有更大的内存容量。

该 64 字节的 CMOS 首先在 IBM PC-XT 机器上用于保存时钟和日期信息。由于这些信息仅用去 14 字节，剩余的字节就用来存放一些系统配置数据了。

CMOS 的地址空间是在基本地址空间之外的。因此其中不包括可执行的代码。它需要使用在端口 70h,71h 使用 IN 和 OUT 指令来访问。为了读取指定偏移位置的字节，首先需要使用 OUT 向端口 70h 发送指定字节的偏移值，然后使用 IN 指令从 71h 端口读取指定的字节信息。

这段程序中(行 70)将欲读取的字节地址或上了一个 80h 值是没有必要的。因为那时的 CMOS 内存容量还没有超过 128 字节，因此或上 80h 的操作是没有任何作用的。之所以会有这样的操作是因为当时 Linus 手头缺乏有关 CMOS 方面的资料，CMOS 中时钟和日期的偏移地址都是他逐步实验出来的，也许在他实验中将偏移地址或上 80h（并且还修改了其它地方）后正好取得了所有正确的结果，因此他的代码中也就有了这步不必要的操作。不过从 1.0 版本之后，该操作就被去除了(可参见 1.0 版内核程序 drivers/block/hd.c 第 42 行起的代码)。

下面是 CMOS 内存信息的一张简表。

表 4.1 CMOS 64 字节信息简表

地址偏移值	内容说明
0x00	当前秒值 （实时钟)
0x01	报警秒值
0x02	当前分钟 （实时钟)
0x03	报警分钟值
0x04	当前小时值 （实时钟)
0x05	报警小时值

0x06	一周中的当前天（实时钟）
0x07	一月中的当日日期（实时钟）
0x08	当前月份（实时钟）
0x09	当前年份（实时钟）
0x0a	RTC 状态寄存器 A
0x0b	RTC 状态寄存器 B
0x0c	RTC 状态寄存器 C
0x0d	RTC 状态寄存器 D
0x0e	POST 诊断状态字节
0x0f	停机状态字节
0x10	磁盘驱动器类型
0x11	保留
0x12	硬盘驱动器类型
0x13	保留
0x14	设备字节
0x15	基本内存（低字节）
0x16	基本内存（高字节）
0x17	扩展内存（低字节）
0x18	扩展内存（高字节）
0x19-0x2d	保留
0x2e	校验和（低字节）
0x2f	校验和（高字节）
0x30	1Mb 以上的扩展内存（低字节）
0x31	1Mb 以上的扩展内存（高字节）
0x32	当前所处世纪值
0x33	信息标志
0x34-0x3f	保留

4.2.3.2 调用 fork()创建新进程

fork 是一个系统调用函数。该系统调用复制当前进程，并在进程表中创建一个与原进程(被称为父进程）几乎完全一样的新表项，并执行同样的代码，但该新进程（这里被称为子进程）拥有自己的数据空间和环境参数。

在父进程中，调用 fork()返回的是子进程的进程标识号 PID，而在子进程中 fork()返回的将是 0 值，这样，虽然此时还是在同样一程序中执行，但已开始叉开，各自执行自己的那段代码。如果 fork()调用失败，则会返回小于 0 的值。如示意图 4.2 所示。

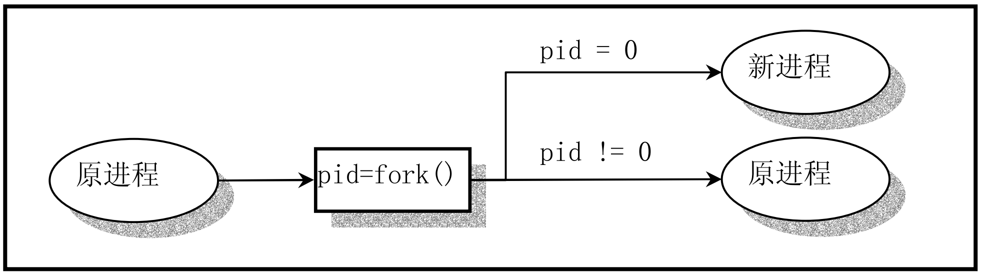


图 4.2 调用 fork()创建新进程

init 程序即是用 fork()调用的返回值来区分和执行不同的代码段的。上面代码中第 179 和 194 行是子进程的判断并开始子进程代码块的执行（利用 execve()系统调用执行其它程序，这里执行的是 sh），第 186 和 202 行是父进程执行的代码块。

4.3 本章小结

对于 0.11 版内核，通过上面代码分析可知，只要根文件系统是一个 MINIX 文件系统，并且其中只要包含文件/etc/rc、/bin/sh、/dev/* 以及一些目录/etc/、/dev/、/bin/、/home/、/home/root/ 就可以构成一个最简单的根文件系统，让 Linux 运行起来。

从这里开始，对于后续章节的阅读，可以将 init.c 程序作为一条主线进行，并不需要按章节顺序阅读。为了能比较顺利地理解以下各章内容，作者强力希望读者此时能再次复习 32 位保护模式运行的机制，详细阅读一下附录中所提供的有关内容，或者参考 Intel 80x86 的有关书籍，把保护模式下的运行机制彻底弄清楚，然后再继续阅读。

第 5 章 内核代码(kernel)

5.1 概述

linux/kernel/目录下共包括 10 个 C 语言文件和 2 个汇编语言文件以及一个 kernel 下编译文件的管理配置文件 Makefile。见列表 5.1 所示。其中三个子目录中代码注释的将放在后续章节中进行。本章主要对这 13 个代码文件进行注释。首先我们对所有程序的基本功能进行概括性地总体介绍，以便一开始就对这 12 个文件所实现的功能和它们之间的相互调用关系有个大致的了解，然后逐一对代码进行详细地注释。

列表 5.1 linux/kernel/**目录**

文件名	大小	最后修改时间(GMT)	描述
blk_drv/		1991-12-08 14:09:29	
chr_drv/		1991-12-08 18:36:09	
math/		1991-12-08 14:09:58	
Makefile	3309 bytes	1991-12-02 03:21:37	m
asm.s	2335 bytes	1991-11-18 00:30:28	m
exit.c	4175 bytes	1991-12-07 15:47:55	m
fork.c	3693 bytes	1991-11-25 15:11:09	m
mktime.c	1461 bytes	1991-10-02 14:16:29	m
panic.c	448 bytes	1991-10-17 14:22:02	m
printk.c	734 bytes	1991-10-02 14:16:29	m
sched.c	8242 bytes	1991-12-04 19:55:28	m
signal.c	2651 bytes	1991-12-07 15:47:55	m
sys.c	3706 bytes	1991-11-25 19:31:13	m
system_call.s	5265 bytes	1991-12-04 13:56:34	m
traps.c	4951 bytes	1991-10-30 20:20:40	m
vsprintf.c	4800 bytes	1991-10-02 14:16:29	m

5.1.1 总体功能描述

该目录下的代码文件从功能上可以分为三类，一类是硬件（异常）中断处理程序文件，一类是系统调用服务处理程序文件，另一类是进程调度等通用功能文件。参见图 1.5。我们现在根据这个分类方式，从实现的功能上进行更详细的说明。

5.1.1.1 硬件中断处理类程序

主要包括两个代码文件：asm.s 和 traps.c 文件。asm.s 用于实现大部分硬件异常所引起的中断的汇编语言处理过程。而 traps.c 程序则实现了 asm.s 的中断处理过程中调用的 c 函数。另外几个硬件中断处理程序在文件 system_call.s 和 mm/page.s 中实现。

中断信号通常可以分为两类：硬件中断和软件中断(异常)。每个中断是由 0-255 之间的一个数字来标识。对于中断 int0--int31(0x00--0x1f)，每个中断的功能是由 Intel 固定设定或保留用的, 属于软件中断,但 Intel 称之为异常。因为是由 CPU 执行指令时探测到异常时引起的。通常还可分为故障(Fault)和陷阱(traps)两类。中断 int32--int255 (0x20--0xff)可以由用户自己设定。在 Linux 系统中，则将 int32--int47(0x20--0x2f)对应于 8259A 中断控制芯片发出的硬件中断请求信号 IRQ0-IRQ15；并把程序编程发出的系统调用(system_call)中断设置为 int128(0x80)。

asm.s 代码文件主要涉及对 Intel 保留中断 int0--int16 的处理，其余保留的中断 int17-int31 由 Intel 公司留作今后扩充使用。对应于中断控制器芯片各 IRQ 发出的 int32-int47 的 16 个处理程序将分别在各种硬件（如时钟、键盘、软盘、数学协处理器、硬盘等）初始化程序中处理。Linux 系统调用中断 int128(0x80)的处理则将在 kernel/system_call.s 中给出。各个中断的具体定义见代码注释后其它信息一节中的说明。

对一个硬件异常所引起的中断的处理过程见下度所示（图 5.1）。

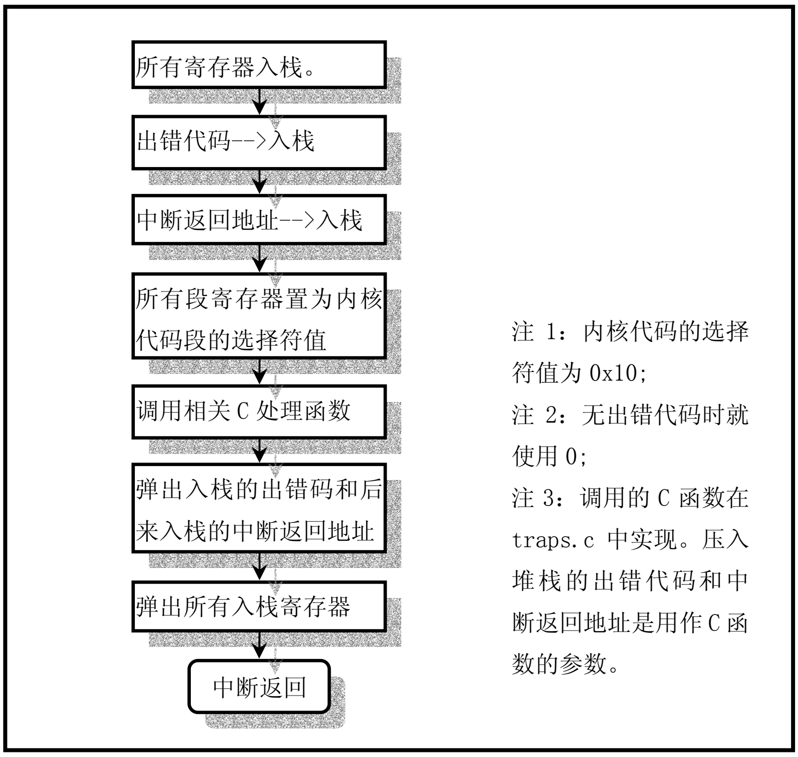


图 5.1 硬件异常（故障、陷阱）所引起的中断处理流程

由于有些异常引起中断时，CPU 内部会产生一个出错代码压入堆栈（异常中断 int 8 和 int10 - int 14），而其它的中断却并不带有这个出错代码（例如被零除出错和边界检查出错等），因此，asm.s 程序中将所有中断的处理根据是否携带出错代码而分别进行处理。但处理流程还是一样的。

会产生出错代码的

5.1.1.2 系统调用处理相关程序

Linux 中应用程序调用内核的功能是通过中断调用 int 0x80 进行的，寄存器 eax 中放调用号。因此该中断调用被称为系统调用。实现系统调用的相关文件包括 system_call.s、fork.c、signal.c、sys.c 和 exit.c 文件。

system_call.s 程序的作用类似于硬件中断处理中的 asm.s 程序的作用，另外还对时钟中断和硬盘、软盘中断进行处理。而 fork.c 和 signal.c 中的一个函数则类似于 traps.c 程序的作用，为系统中断调用提供 C 处理函数。fork.c 程序提供两个 C 处理函数：find_empty_process()和 copy_process()。

signal.c 程序还提供一个处理有关进程信号的函数 do_signal()，在系统调用中断处理过程中被调用。另外还包括 4 个系统调用 sys_xxx()函数。

sys.c 和 exit.c 程序实现了其它一些 sys_xxx()系统调用函数。这些 sys_xxx()函数都是相应系统调用所需调用的处理函数，有些是使用汇编语言实现的，如 sys_execve()；而另外一些则用 C 语言实现（例如 signal.c 中的 4 个系统调用函数）。

我们可以根据这些函数的简单命名规则这样来理解：通常以'do_'开头的中断处理过程中调用的 C 函数，要么是系统调用处理过程中通用的函数，要么是某个系统调用专用的；而以'sys_'开头的系统调用函数则是指定的系统调用的专用处理函数。例如，do_signal()函数基本上是所有系统调用都要执行的函数，而 do_hd()、do_execve()则是某个系统调用专用的 C 处理函数。

5.1.1.3 其它通用类程序

这些程序包括 schedule.c、mktime.c、panic.c、printk.c 和 vsprintf.c。

schedule.c 程序包括内核调用最频繁的 schedule()、sleep_on()和 wakeup()函数，是内核的核心调度程序，用于对进程的执行进行切换或改变进程的执行状态。mktime.c 程序中仅包含一个内核使用的时间函数 mktime()，仅在 init/main.c 中被调用一次。panic.c 中包含一个 panic()函数，用于在内核运行出现错误时显示出错信息并停机。printk.c 和 vsprintf.c 是内核显示信息的支持程序，实现了内核专用显示函数 printk()和字符串格式化输出函数 vsprintf()。

5.2 Makefile 文件

5.2.1 功能简介

编译 linux/kernel/下程序的 make 配置文件，不包括三个子目录。该文件的组成格式与第一章中列表 1.2 的基本相同，在阅读时可以参考列表 1.2 中的有关注释。

5.2.2 文件注释

列表 5.2 linux/kernel/Makefile 文件

```
#
# Makefile for the FREAX-kernel.
#
# Note! Dependencies are done automagically by 'make dep', which also
# removes any old dependencies. DON'T put your own dependencies here
# unless it's something special (ie not a .c file).
#
# FREAX 内核的 Makefile 文件。
#
# 注意！依赖关系是由'make dep'自动进行的，它也会自动去除原来的依赖信息。不要把你自己的
# 依赖关系信息放在这里，除非是特别文件的（也即不是一个.c 文件的信息）。
# (Linux 最初的名字叫 FREAX，后来被 ftp.funet.fi 的管理员改成 Linux 这个名字)

AR      =gar     # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。
AS      =gas     # GNU 的汇编程序。
LD      =gld     # GNU 的连接程序。
LDFLAGS =-s -x   # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
CC      =gcc     # GNU C 语言编译器。
```

```
14 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
15         -finline-functions -mstring-insns -nostdinc -I../include
    # C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
    # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
    # 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
    # 单短小的函数代码嵌入调用程序中；-mstring-insns Linus 自己填加的优化选项，以后不再使用；
    # -nostdinc -I../include 不使用默认路径中的包含文件，而使用这里指定目录中的(../include)。
16 CPP     =gcc -E -nostdinc -I../include
    # C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
    # 出设备或指定的输出文件中；-nostdinc -I../include 同前。
17
    # 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
    # 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
    # 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
    # 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。其中$*.s（或$@）是自动目标变量，
    # $<代表第一个先决条件，这里即是符合条件*.c 的文件。
18 .c.s:
19         $(CC) $(CFLAGS) \
20         -S -o $*.s $<
  # 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。22 行是实现该操作的具体命令。
21 .s.o:
22         $(AS) -c -o $*.o $<
23 .c.o:                     # 类似上面，*.c 文件-→*.o 目标文件。不进行连接。
24         $(CC) $(CFLAGS) \
25         -c -o $*.o $<
26
27 OBJS  = sched.o system_call.o traps.o asm.o fork.o \   # 定义目标文件变量 OBJS。
28         panic.o printk.o vsprintf.o sys.o exit.o \
29         signal.o mktime.o
30
31 kernel.o: $(OBJS)         # 在有了先决条件 OBJS 后使用下面的命令连接成目标 kernel.o
32         $(LD) -r -o kernel.o $(OBJS)
33         sync
34
    # 下面的规则用于清理工作。当执行'make clean'时，就会执行 36--40 行上的命令，去除所有编译
    # 连接生成的文件。'rm'是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
35 clean:
36         rm -f core *.o *.a tmp_make keyboard.s
37         for i in *.c;do rm -f `basename $$i .c`.s;done
38         (cd chr_drv; make clean)    # 进入 chr_drv/目录；执行该目录 Makefile 中的 clean 规则。
39         (cd blk_drv; make clean)
40         (cd math; make clean)
41
    # 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
    # 使用字符串编辑程序 sed 对 Makefile 文件（这里即是自己）进行处理，输出为删除 Makefile
    # 文件中'### Dependencies'行后面的所有行（下面从 51 开始的行），并生成 tmp_make
    # 临时文件（43 行的作用）。然后对 kernel/目录下的每一个 C 文件执行 gcc 预处理操作.
    # -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
    # 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
    # 文件名加上其依赖关系--该源文件中包含的所有头文件列表。把预处理结果都添加到临时
    # 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。
42 dep:
43         sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
```

```
	(for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,'`" "; \
		$(CPP) -M $$i;done) >> tmp_make
	cp tmp_make Makefile
	(cd chr_drv; make dep)     # 对 chr_drv/目录下的 Makefile 文件也作同样的处理。
	(cd blk_drv; make dep)

### Dependencies:
exit.s exit.o : exit.c ../include/errno.h ../include/signal.h \
  ../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
  ../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h \
  ../include/asm/segment.h
fork.s fork.o : fork.c ../include/errno.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
  ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
  ../include/asm/segment.h ../include/asm/system.h
mktime.s mktime.o : mktime.c ../include/time.h
panic.s panic.o : panic.c ../include/linux/kernel.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
  ../include/linux/mm.h ../include/signal.h
printk.s printk.o : printk.c ../include/stdarg.h ../include/stddef.h \
  ../include/linux/kernel.h
sched.s sched.o : sched.c ../include/linux/sched.h ../include/linux/head.h \
  ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
  ../include/signal.h ../include/linux/kernel.h ../include/linux/sys.h \
  ../include/linux/fdreg.h ../include/asm/system.h ../include/asm/io.h \
  ../include/asm/segment.h
signal.s signal.o : signal.c ../include/linux/sched.h ../include/linux/head.h \
  ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
  ../include/signal.h ../include/linux/kernel.h ../include/asm/segment.h
sys.s sys.o : sys.c ../include/errno.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
  ../include/linux/mm.h ../include/signal.h ../include/linux/tty.h \
  ../include/termios.h ../include/linux/kernel.h ../include/asm/segment.h \
  ../include/sys/times.h ../include/sys/utsname.h
traps.s traps.o : traps.c ../include/string.h ../include/linux/head.h \
  ../include/linux/sched.h ../include/linux/fs.h ../include/sys/types.h \
  ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
  ../include/asm/system.h ../include/asm/segment.h ../include/asm/io.h
vsprintf.s vsprintf.o : vsprintf.c ../include/stdarg.h ../include/string.h
```

5.3 asm.s 文件

5.3.1 功能描述

asm.s 汇编程序中包括大部分 CPU 探测到的异常故障处理的底层代码，也包括数学协处理器（FPU）的异常处理。该程序与 kernel/traps.c 程序有着密切的关系。该程序的主要处理方式是在中断处理程序中调用相应的 C 函数程序，显示出错位置和出错号，然后退出中断。

在阅读这段代码时参照下面堆栈变化示意图将是很有帮助的（图中每个行代表 4 个字节）。在开始执行程序之前，堆栈指针 esp 指在中断返回地址一栏（图中 esp0 处）。当把将要调用的 C 函数 do_divide_error()或其它 C 函数地址入栈后，指针位置是 esp1 处,此时通过交换指令，该函数的地址被放入 eax 寄存器中，而原来 eax 的值被保存到堆栈上。在把一些寄存器入栈后，堆栈指针位置在 esp2 处。当正式调用 do_divide_error()之前，程序将开始执行时的 esp0 堆栈指针值压入堆栈，放到了 esp3 处，并在中断返回弹出入栈的寄存器之前指针通过加上 8 又回到 esp2 处。

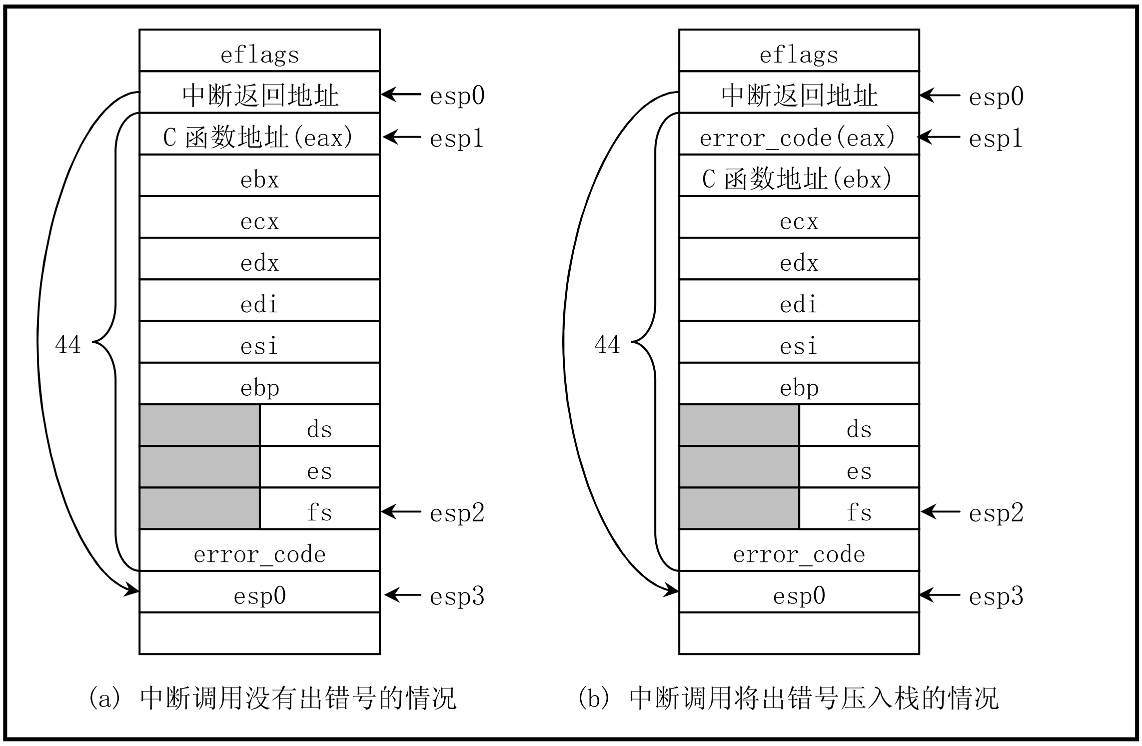


图 5.2 出错处理堆栈变化示意图

正式调用 do_divide_error()之前把出错代码以及 esp0 入栈的原因是为了作为调用 C 函数 do_divide_error()的参数。在 traps.c 中该函数的原形为：

void do_divide_error(long esp, long error_code)。

因此在这个 C 函数中就可以打印出出错的位置和错误号。程序中其余异常出错的处理过程与这里描述的过程基本类似。

系统调用处理过程的整个流程见图 5.3 所示。

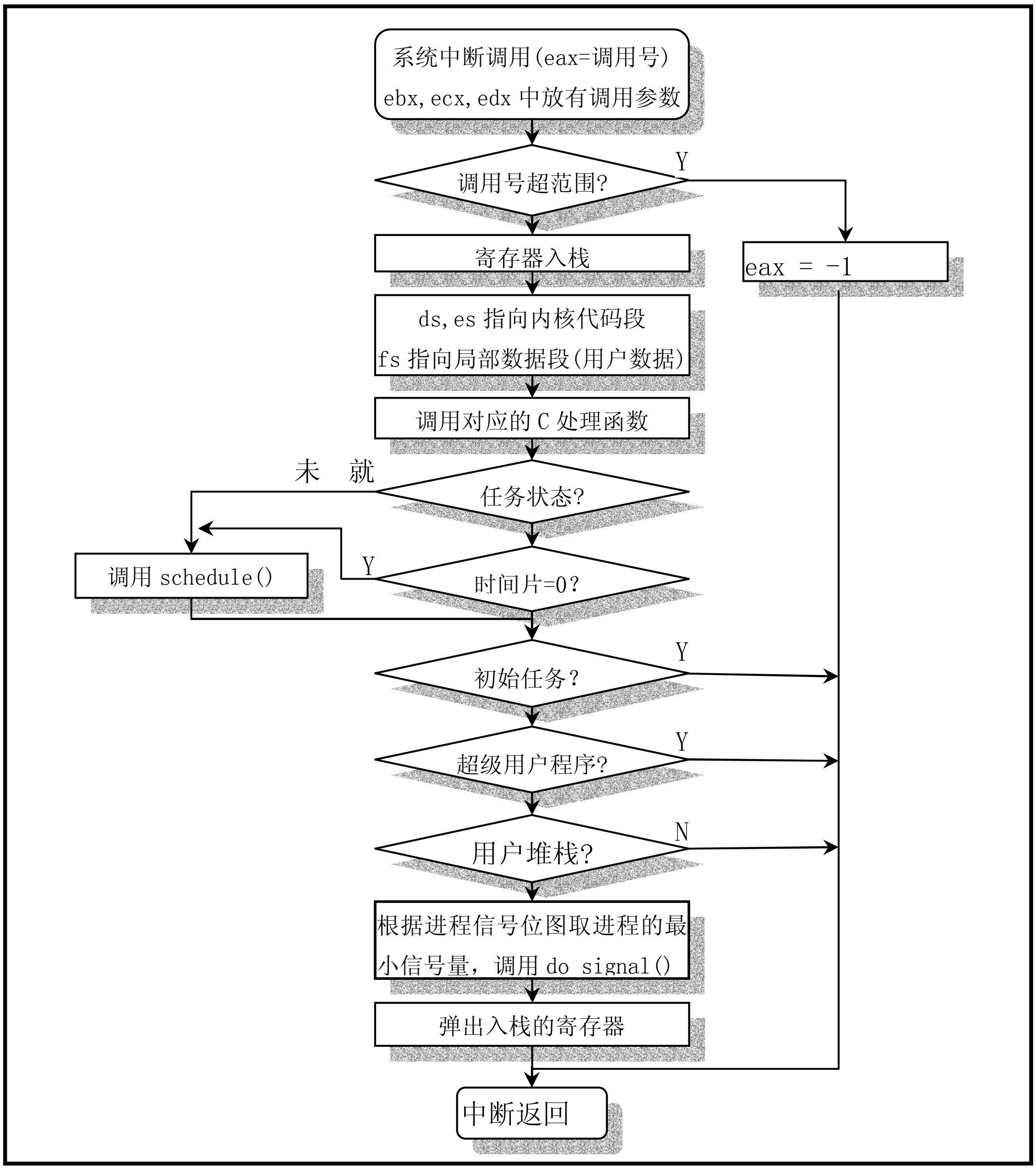


图 5.3 系统中断调用处理流程

5.3.2 代码注释

列表 5.3 linux/kernel/asm.s 程序

```
/*
 *  linux/kernel/asm.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * asm.s contains the low-level code for most hardware faults.
 * page_exception is handled by the mm, so that isn't here. This
 * file also handles (hopefully) fpu-exceptions due to TS-bit, as
 * the fpu must be properly saved/resored. This hasn't been tested.
```

```
12  */
/*
 * asm.s 程序中包括大部分的硬件故障（或出错）处理的底层次代码。页异常是由内存管理程序
 * mm 处理的，所以不在这里。此程序还处理（希望是这样）由于 TS-位而造成的 fpu 异常，
 * 因为 fpu 必须正确地进行保存/恢复处理，这些还没有测试过。
 */
13
 # 本代码文件主要涉及对 Intel 保留的中断 int0--int16 的处理（int17-int31 留作今后使用）。
 # 以下是一些全局函数名的声明，其原形在 traps.c 中说明。
14 .globl _divide_error,_debug,_nmi,_int3,_overflow,_bounds,_invalid_op
15 .globl _double_fault,_coprocessor_segment_overrun
16 .globl _invalid_TSS,_segment_not_present,_stack_segment
17 .globl _general_protection,_coprocessor_error,_irq13,_reserved
18
# int0 -- （下面这段代码的含义参见图 4.1(a)）。
# 下面是被零除出错(divide_error)处理代码。标号'_divide_error'实际上是 C 语言函
# 数 divide_error()编译后所生成模块中对应的名称。'_do_divide_error'函数在 traps.c 中。
19 _divide_error:
20         pushl $_do_divide_error # 首先把将要调用的函数地址入栈。这段程序的出错号为 0。
21 no_error_code:                  # 这里是无出错号处理的入口处，见下面第 55 行等。
22         xchgl %eax,(%esp)       # _do_divide_error 的地址 ➔ eax，eax 被交换入栈。
23         pushl %ebx
24         pushl %ecx
25         pushl %edx
26         pushl %edi
27         pushl %esi
28         pushl %ebp
29         push %ds                # ！！16 位的段寄存器入栈后也要占用 4 个字节。
30         push %es
31         push %fs
32         pushl $0                # "error code"   # 将出错码入栈。
33         lea 44(%esp),%edx       # 取原调用返回地址处堆栈指针位置，并压入堆栈。
34         pushl %edx
35         movl $0x10,%edx         # 内核代码数据段选择符。
36         mov %dx,%ds
37         mov %dx,%es
38         mov %dx,%fs
39         call *%eax              # 调用 C 函数 do_divide_error()。
40         addl $8,%esp            # 让堆栈指针重新指向寄存器 fs 入栈处。
41         pop %fs
42         pop %es
43         pop %ds
44         popl %ebp
45         popl %esi
46         popl %edi
47         popl %edx
48         popl %ecx
49         popl %ebx
50         popl %eax               # 弹出原来 eax 中的内容。
51         iret
52
# int1 -- debug 调试中断入口点。处理过程同上。
53 _debug:
```

```
        pushl $_do_int3          # _do_debug C 函数指针入栈。以下同。
        jmp no_error_code

# int2 -- 非屏蔽中断调用入口点。
_nmi:
        pushl $_do_nmi
        jmp no_error_code

# int3 -- 同_debug。
_int3:
        pushl $_do_int3
        jmp no_error_code

# int4 -- 溢出出错处理中断入口点。
_overflow:
        pushl $_do_overflow
        jmp no_error_code

# int5 -- 边界检查出错中断入口点。
_bounds:
        pushl $_do_bounds
        jmp no_error_code

# int6 -- 无效操作指令出错中断入口点。
_invalid_op:
        pushl $_do_invalid_op
        jmp no_error_code

# int9 -- 协处理器段超出出错中断入口点。
_coprocessor_segment_overrun:
        pushl $_do_coprocessor_segment_overrun
        jmp no_error_code

# int15 - 保留。
_reserved:
        pushl $_do_reserved
        jmp no_error_code

# int45 -- ( = 0x20 + 13 ) 数学协处理器发出的中断。
_irq13:
        pushl %eax
        xorb %al,%al
        outb %al,$0xF0           # 通过写 0xF0 端口，本中断将消除 CPU 的 BUSY 延续信号，并重新
                                 # 激活 80387 的处理器扩展请求引脚 PEREQ。该操作主要是为了确保
                                 # 在继续执行 80387 的任何指令之前，响应本中断。
        movb $0x20,%al
        outb %al,$0x20           # 向 8259 主中断控制芯片发送 EOI（中断结束）信号。
        jmp 1f                   # 这两个跳转指令起延时作用。
1:      jmp 1f
1:      outb %al,$0xA0           # 再向 8259 从中断控制芯片发送 EOI（中断结束）信号。
        popl %eax
        jmp _coprocessor_error   # _coprocessor_error 原来在本文件中，现在已经放到
                                 # （kernel/system_call.s, 131）
```

```
# 以下中断在调用时会在中断返回地址之后将出错号压入堆栈，因此返回时也需要将出错号弹出。
# int8 -- 双出错故障。（下面这段代码的含义参见图 4.1(b)）。
_double_fault:
        pushl $_do_double_fault         # C 函数地址入栈。
error_code:
        xchgl %eax,4(%esp)              # error code <-> %eax
        xchgl %ebx,(%esp)               # &function <-> %ebx
        pushl %ecx
        pushl %edx
        pushl %edi
        pushl %esi
        pushl %ebp
        push %ds
        push %es
        push %fs
        pushl %eax                      # error code    # 出错号入栈。
        lea 44(%esp),%eax               # offset        # 程序返回地址处堆栈指针位置值入栈。
        pushl %eax
        movl $0x10,%eax         # 置内核数据段选择符。
        mov %ax,%ds
        mov %ax,%es
        mov %ax,%fs
        call *%ebx              # 调用相应的 C 函数，其参数已入栈。
        addl $8,%esp            # 堆栈指针重新指向栈中放置 fs 内容的位置。
        pop %fs
        pop %es
        pop %ds
        popl %ebp
        popl %esi
        popl %edi
        popl %edx
        popl %ecx
        popl %ebx               # 弹出 C 函数地址。
        popl %eax               # 弹出出错号。
        iret

# int10 -- 无效的任务状态段(TSS)。
_invalid_TSS:
        pushl $_do_invalid_TSS
        jmp error_code

# int11 -- 段不存在。
_segment_not_present:
        pushl $_do_segment_not_present
        jmp error_code

# int12 -- 堆栈段错误。
_stack_segment:
        pushl $_do_stack_segment
        jmp error_code

# int13 -- 一般保护性出错。
```

```
_general_protection:
        pushl $_do_general_protection
        jmp error_code

# int7 -- 设备不存在(_device_not_available)在(kernel/system_call.s,148)
# int14 -- 页错误(_page_fault)在(mm/page.s,14)
# int16 -- 协处理器错误(_coprocessor_error)在(kernel/system_call.s,131)
# 时钟中断 int 0x20 (_timer_interrupt)在(kernel/system_call.s,176)
# 系统调用 int 0x80 (_system_call)在（kernel/system_call.s,80）
```

5.3.3 其它信息

5.3.3.1 Intel 保留中断向量的定义

这里给出了 Intel 保留中断向量具体含义的说明，见表 5.1 所示。

表 5.1 Intel 保留的中断号含义

中断号	名称	类型	信号	说明
0	Devide error	故障	SIGFPE	当进行除以零的操作时产生。
1	Debug	陷阱 故障	SIGTRAP	当进行程序单步跟踪调试时，设置了标志寄存器 eflags 的 T 标志时产生这个中断。
2	nmi	硬件		由不可屏蔽中断 NMI 产生。
3	Breakpoint	陷阱	SIGTRAP	由断点指令 int3 产生，与 debug 处理相同。
4	Overflow	陷阱	SIGSEGV	eflags 的溢出标志 OF 引起。
5	Bounds check	故障	SIGSEGV	寻址到有效地址以外时引起。
6	Invalid Opcode	故障	SIGILL	CPU 执行时发现一个无效的指令操作码。
7	Device not available	故障	SIGSEGV	设备不存在，指协处理器。在两种情况下会产生该中断：(a)CPU 遇到一个转意指令并且 EM 置位时。在这种情况下处理程序应该模拟导致异常的指令。(b)MP 和 TS 都在置位状态时，CPU 遇到 WAIT 或一个转意指令。在这种情况下，处理程序在必要时应该更新协处理器的状态。
8	Double fault	故障	SIGSEGV	双故障出错。
9	Coprocessor segment overrun	故障	SIGFPE	协处理器段超出。
10	Invalid TSS	故障	SIGSEGV	CPU 切换时发觉 TSS 无效。
11	Segment not present	故障	SIGBUS	描述符所指的段不存在。
12	Stack segment	故障	SIGBUS	堆栈段不存在或寻址越出堆栈段。
13	General protection	故障	SIGSEGV	没有符合 80386 保护机制（特权级）的操作引起。
14	Page fault	故障	SIGSEGV	页不在内存。
15	Reserved			
16	Coprocessor error	故障	SIGFPE	协处理器发出的发出的出错信号引起。

5.4 traps.c 程序

5.4.1 功能描述

traps.c 程序主要包括一些在处理异常故障（硬件中断）的底层代码 asm.s 中调用的相应 C 函数。用于显示出错位置和出错号等调试信息。其中的 die()通用函数用于在中断处理中显示详细的出错信息，而代码最后的初始化函数 trap_init()是在前面 init/main.c 中被调用，用于硬件异常处理中断向量（陷阱门）的初始化，并设置允许中断请求信号的到来。在阅读本程序时需要参考 asm.s 程序。

5.4.2 代码注释

列表 5.4 linux/kernel/traps.c 程序

```
/*
 *  linux/kernel/traps.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 'Traps.c' handles hardware traps and faults after we have saved some
 * state in 'asm.s'. Currently mostly a debugging-aid, will be extended
 * to mainly kill the offending process (probably by giving it a signal,
 * but possibly by killing it outright if necessary).
 */
/*
 * 在程序 asm.s 中保存了一些状态后，本程序用来处理硬件陷阱和故障。目前主要用于调试目的，
 * 以后将扩展用来杀死遭损坏的进程（主要是通过发送一个信号，但如果必要也会直接杀死）。
 */
#include <string.h>          // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。

#include <linux/head.h>      // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
#include <linux/sched.h>     // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/system.h>      // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
#include <asm/segment.h>     // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
#include <asm/io.h>          // 输入/输出头文件。定义硬件端口输入/输出宏汇编语句。

// 以下语句定义了三个嵌入式汇编宏语句函数。有关嵌入式汇编的基本语法见列表后或参见附录。
// 取段 seg 中地址 addr 处的一个字节。
#define get_seg_byte(seg,addr) ({ \
register char __res; \
__asm__("push %%fs;mov %%ax,%%fs;movb %%fs:%2,%%al;pop %%fs" \
        :"=a" (__res):"" (seg),"m" (*(addr))); \
__res;})

// 取段 seg 中地址 addr 处的一个长字（4 字节）。
#define get_seg_long(seg,addr) ({ \
register unsigned long __res; \
```

```
__asm__("push %%fs;mov %%ax,%%fs;movl %%fs:%2,%%eax;pop %%fs" \
        :"=a" (__res):"" (seg),"m" (*(addr))); \
__res;})

// 取 fs 段寄存器的值（选择符）。
#define _fs() ({ \
register unsigned short __res; \
__asm__("mov %%fs,%%ax":"=a" (__res):); \
__res;})

// 以下定义了一些函数原型。
int do_exit(long code);                    // (kernel/exit.c,102)

void page_exception(void);                 // [??]

// 以下定义了一些中断处理程序原型，代码在（kernel/asm.s 或 system_call.s）中。
void divide_error(void);                   // int0 (kernel/asm.s,19)。
void debug(void);                          // int1 (kernel/asm.s,53)。
void nmi(void);                            // int2 (kernel/asm.s,57)。
void int3(void);                           // int3 (kernel/asm.s,61)。
void overflow(void);                       // int4 (kernel/asm.s,65)。
void bounds(void);                         // int5 (kernel/asm.s,69)。
void invalid_op(void);                     // int6 (kernel/asm.s,73)。
void device_not_available(void);           // int7 (kernel/system_call.s,148)。
void double_fault(void);                   // int8 (kernel/asm.s,97)。
void coprocessor_segment_overrun(void);    // int9 (kernel/asm.s,77)。
void invalid_TSS(void);                    // int10 (kernel/asm.s,131)。
void segment_not_present(void);            // int11 (kernel/asm.s,135)。
void stack_segment(void);                  // int12 (kernel/asm.s,139)。
void general_protection(void);             // int13 (kernel/asm.s,143)。
void page_fault(void);                     // int14 (mm/page.s,14)。
void coprocessor_error(void);              // int16 (kernel/system_call.s,131)。
void reserved(void);                       // int15 (kernel/asm.s,81)。
void parallel_interrupt(void);             // int39 (kernel/system_call.s,280)。
void irq13(void);                          // int45 协处理器中断处理(kernel/asm.s,85)。

// 该子程序用来打印出错中断的名称、出错号、调用程序的 EIP、EFLAGS、ESP、fs 段寄存器值、
// 段的基址、段的长度、进程号 pid、任务号、10 字节指令码。如果堆栈在用户数据段，则还
// 打印 16 字节的堆栈内容。
static void die(char * str,long esp_ptr,long nr)
{
        long * esp = (long *) esp_ptr;
        int i;

        printk("%s: %04x\n\r",str,nr&0xffff);
        printk("EIP:\t%04x:%p\nEFLAGS:\t%p\nESP:\t%04x:%p\n",
                esp[1],esp[0],esp[2],esp[4],esp[3]);
        printk("fs: %04x\n",_fs());
        printk("base: %p, limit: %p\n",get_base(current->ldt[1]),get_limit(0x17));
        if (esp[4] == 0x17) {
                printk("Stack: ");
                for (i=0;i<4;i++)
                        printk("%p ",get_seg_long(0x17,i+(long *)esp[3]));
```

```
                        printk("\n");
                }
                str(i);
                printk("Pid: %d, process nr: %d\n\r",current->pid,0xffff & i);
                for(i=0;i<10;i++)
                        printk("%02x ",0xff & get_seg_byte(esp[1],(i+(char *)esp[0])));
                printk("\n\r");
                do_exit(11);            /* play segment exception */
}

// 以下这些以 do_开头的函数是对应名称中断处理程序调用的 C 函数。
void do_double_fault(long esp, long error_code)
{
                die("double fault",esp,error_code);
}

void do_general_protection(long esp, long error_code)
{
                die("general protection",esp,error_code);
}

void do_divide_error(long esp, long error_code)
{
                die("divide error",esp,error_code);
}

void do_int3(long * esp, long error_code,
                long fs,long es,long ds,
                long ebp,long esi,long edi,
                long edx,long ecx,long ebx,long eax)
{
                int tr;

                __asm__("str %%ax":"=a" (tr):"" (0));    // 取任务寄存器值➔tr。
                printk("eax\t\tebx\t\tecx\t\tedx\n\r%8x\t%8x\t%8x\t%8x\n\r",
                        eax,ebx,ecx,edx);
                printk("esi\t\tedi\t\tebp\t\tesp\n\r%8x\t%8x\t%8x\t%8x\n\r",
                        esi,edi,ebp,(long) esp);
                printk("\n\rds\tes\tfs\ttr\n\r%4x\t%4x\t%4x\t%4x\n\r",
                        ds,es,fs,tr);
                printk("EIP: %8x   CS: %4x  EFLAGS: %8x\n\r",esp[0],esp[1],esp[2]);
}

void do_nmi(long esp, long error_code)
{
                die("nmi",esp,error_code);
}

void do_debug(long esp, long error_code)
{
                die("debug",esp,error_code);
}

```

```c
void do_overflow(long esp, long error_code)
{
        die("overflow",esp,error_code);
}

void do_bounds(long esp, long error_code)
{
        die("bounds",esp,error_code);
}

void do_invalid_op(long esp, long error_code)
{
        die("invalid operand",esp,error_code);
}

void do_device_not_available(long esp, long error_code)
{
        die("device not available",esp,error_code);
}

void do_coprocessor_segment_overrun(long esp, long error_code)
{
        die("coprocessor segment overrun",esp,error_code);
}

void do_invalid_TSS(long esp,long error_code)
{
        die("invalid TSS",esp,error_code);
}

void do_segment_not_present(long esp,long error_code)
{
        die("segment not present",esp,error_code);
}

void do_stack_segment(long esp,long error_code)
{
        die("stack segment",esp,error_code);
}

void do_coprocessor_error(long esp, long error_code)
{
        if (last_task_used_math != current)
                return;
        die("coprocessor error",esp,error_code);
}

void do_reserved(long esp, long error_code)
{
        die("reserved (15,17-47) error",esp,error_code);
}

// 下面是异常（陷阱）中断程序初始化子程序。设置它们的中断调用门（中断向量）。
```

```
// set_trap_gate()与 set_system_gate()的主要区别在于前者设置的特权级为 0，后者是 3。因此
// 断点陷阱中断 int3、溢出中断 overflow 和边界出错中断 bounds 可以由任何程序产生。
// 这两个函数均是嵌入式汇编宏程序(include/asm/system.h,第 36 行、39 行)。
void trap_init(void)
{
        int i;

        set_trap_gate(0,&divide_error);  // 设置除操作出错的中断向量值。以下雷同。
        set_trap_gate(1,&debug);
        set_trap_gate(2,&nmi);
        set_system_gate(3,&int3);       /* int3-5 can be called from all */
        set_system_gate(4,&overflow);
        set_system_gate(5,&bounds);
        set_trap_gate(6,&invalid_op);
        set_trap_gate(7,&device_not_available);
        set_trap_gate(8,&double_fault);
        set_trap_gate(9,&coprocessor_segment_overrun);
        set_trap_gate(10,&invalid_TSS);
        set_trap_gate(11,&segment_not_present);
        set_trap_gate(12,&stack_segment);
        set_trap_gate(13,&general_protection);
        set_trap_gate(14,&page_fault);
        set_trap_gate(15,&reserved);
        set_trap_gate(16,&coprocessor_error);
        for (i=17;i<48;i++)
                set_trap_gate(i,&reserved);  // 将 int17-48 的陷阱门均设置为 reserved。
        set_trap_gate(45,&irq13);
        outb_p(inb_p(0x21)&0xfb,0x21);       // 允许主 8259A 芯片的 IRQ2 中断请求。
        outb(inb_p(0xA1)&0xdf,0xA1);         // 允许从 8259A 芯片的 IRQ13 中断请求。
        set_trap_gate(39,&parallel_interrupt);
}
```

5.4.3 其它信息

5.4.3.1 嵌入式汇编的基本格式

本节是第一次在内核源程序中接触到 C 语言中的嵌入式汇编代码。由于我们在通常的 C 语言程序的编制过程中一般是不会使用嵌入式汇编程序的，因此这里有必要对其基本格式进行简单的描述，详细的说明可参见 GNU gcc 手册中[5]第 4 章的内容（Extensions to the C Language Family），或见参考文献[20]（Using Inline Assembly with gcc）。

具有输入和输出参数的嵌入汇编的基本格式为：

```
asm(“汇编语句”
    : 输出寄存器
    : 输入寄存器
    : 会被修改的寄存器 );
```

其中，“汇编语句”是你写汇编指令的地方；“输出寄存器”表示当这段嵌入汇编执行完之后，哪些

寄存器用于存放输出数据。此地，这些寄存器会分别对应一 C 语言表达式或一个内存地址；“输入寄存器”表示在开始执行汇编代码时，这里指定的一些寄存器中应存放的输入值，它们也分别对应着一 C 变量或常数值。下面我们用例子来说明嵌入汇编语句的使用方法。

我们在下面列出了前面代码中第 22 行开始的一段代码作为例子来详细解说，为了能看清楚我们将这段代码进行了重新编排和编号。

```
01  #define get_seg_byte(seg,addr) \
02  ({ \
03  register char __res; \
04  __asm__("push %%fs;  \
05            mov %%ax,%%fs;  \
06            movb %%fs:%2,%%al;  \
07            pop %%fs"  \
08            :"=a" (__res)  \
09            :"" (seg),"m" (*(addr))); \
10  __res;})
```

这段 10 行代码定义了一个嵌入汇编语言宏函数。因为是宏语句，需要在一行上定义，因此这里使用反斜杠'\'将这些语句连成一行。这条宏定义将被替换到宏名称在程序中被引用的地方。第 1 行定义了宏的名称，也即是宏函数名称 get_seg_byte(seg,addr)。第 3 行定义了一个寄存器变量__res。第 4 行上的__asm__表示嵌入汇编语句的开始。从第 4 行到第 7 行的 4 条 AT&T 格式的汇编语句。

第 8 行即是输出寄存器，这句的含义是在这段代码运行结束后将 eax 所代表的寄存器的值放入__res 变量中，作为本函数的输出值，"=a"中的"a"称为加载代码，"="表示这是输出寄存器。第 9 行表示在这段代码开始运行时将 seg 放到 eax 寄存器中，""表示使用与上面同个位置的输出相同的寄存器。而(*(addr))表示一个内存偏移地址值。为了在上面汇编语句中使用该地址值，嵌入汇编程序规定把输出和输入寄存器统一按顺序编号，顺序是从输出寄存器序列从左到右从上到下以"%0"开始，分别记为%0、%1、…%9。因此，输出寄存器的编号是%0（这里只有一个输出寄存器），输入寄存器前一部分("" (seg))的编号是%1，而后部分的编号是%2。上面第 6 行上的%2 即代表(*(addr))这个内存偏移量。

现在我们来研究 4—7 行上的代码的作用。第一句将 fs 段寄存器的内容入栈；第二句将 eax 中的段值赋给 fs 段寄存器；第三句是把 fs:(*(addr))所指定的字节放入 al 寄存器中。当执行完汇编语句后，输出寄存器 eax 的值将被放入__res，作为该宏函数的返回值。很简单，不是吗？

通过上面分析，我们知道，宏名称中的 seg 代表一指定的内存段值，而 addr 表示一内存偏移地址量。到现在为止，我们应该很清楚这段程序的功能了吧！该宏函数的功能是从指定段和偏移值的内存地址处取一个字节。

在看下一个例子。

```
01  asm("cld\n\t"
02      "rep\n\t"
03      "stol"
04      : /* 没有输出寄存器 */
05      : "c"(count-1), "a"(fill_value), "D"(dest)
06      : "%ecx", "%edi");
```

1-3 行这三句是通常的汇编语句，用以清方向位，重复保存值。第 4 行说明这段嵌入汇编程序没有

用到输出寄存器。第 5 行的含义是：将 count-1 的值加载到 ecx 寄存器中（加载代码是"c"），fill_value 加载到 eax 中，dest 放到 edi 中。为什么要让 gcc 编译程序去做这样的寄存器值的加载，而不让我们自己做呢？因为 gcc 在它进行寄存器分配时可以进行某些优化工作。例如 fill_value 值可能已经在 eax 中。如果是在一个循环语句中的话，gcc 就可能在整个循环操作中保留 eax，这样就可以在每次循环中少用一个 movl 语句。

最后一行的作用是告诉 gcc 这些寄存器中的值已经改变了。很古怪吧？不过在 gcc 知道你拿这些寄存器做些什么后，这确实能够对 gcc 的优化操作有所帮助。

下面列表中，是一些你可能会用到的寄存器加载代码及其具体的含义。

表 5.2 常用寄存器加载代码说明

代码	说明	代码	说明
a	使用寄存器 eax	m	使用内存地址
b	使用寄存器 ebx	o	使用内存地址并可以加偏移值
c	使用寄存器 ecx	I	使用常数 0-31
d	使用寄存器 edx	J	使用常数 0-63
S	使用 esi	K	使用常数 0-255
D	使用 edi	L	使用常数 0-65535
q	使用动态分配字节可寻址寄存器（eax、ebx、ecx 或 edx）	M	使用常数 0-3
r	使用任意动态分配的寄存器	N	使用 1 字节常数（0-255）
g	使用通用有效的地址即可（eax、ebx、ecx、edx 或内存变量）	O	使用常数 0-31
A	使用 eax 与 edx 联合(64 位)		

下面的例子不是让你自己指定哪个变量使用哪个寄存器，而是让 gcc 为你选择。

```
asm("leal (%1, %1, 4), %0"
      : "=r"(y)
      : "0"(x));
```

第一句汇编语句 leal (r1, r2,4), r3 语句表示 r1+r2*4 ➔ r3。这个例子可以非常快地将 x 乘 5。

其中"%0","%1"是指 gcc 自动分配的寄存器。这里"%1"代表输入值 x 要放入的寄存器，"%0"表示输出值寄存器。输出寄存器代码前一定要加等于号。如果输入寄存器的代码是 0 或为空时，则说明使用与相应输出一样的寄存器。所以，如果 gcc 将 r 指定为 eax 的话，那么上面汇编语句的含义即为：

```
"leal (eax,eax,4), eax"
```

注意：在执行代码时，如果不希望汇编语句被 gcc 优化而挪动地方，就需要在 asm 符号后面添加 volatile 关键词：

```
asm volatile (……);
```

或者更详细的说明为：

```
__asm__ __volatile__ (……);
```

下面在具一个较长的例子，如果能看得懂，那就说明嵌入汇编代码对你来说基本没问题了。这段代

码是从 include/string.h 文件中摘取的，是 strncmp()字符串比较函数的一种实现。需要注意的是，其中每行中的"\n\t"是用于 gcc 预处理程序输出列表好看而设置的，含义与 C 语言中相同。

```
    //// 字符串 1 与字符串 2 的前 count 个字符进行比较。
    // 参数：cs - 字符串 1，ct - 字符串 2，count - 比较的字符数。
    // %0 - eax(__res)返回值，%1 - edi(cs)串 1 指针，%2 - esi(ct)串 2 指针，%3 - ecx(count)。
    // 返回：如果串 1 > 串 2，则返回 1；串 1 = 串 2，则返回 0；串 1 < 串 2，则返回-1。
extern inline int strncmp(const char * cs,const char * ct,int count)
{
register int __res ;                 // __res 是寄存器变量。
__asm__("cld\n"                      // 清方向位。
        "1:\tdecl %3\n\t"            // count--。
        "js 2f\n\t"                  // 如果 count<0，则向前跳转到标号 2。
        "lodsb\n\t"                  // 取串 2 的字符 ds:[esi]➔al，并且 esi++。
        "scasb\n\t"                  // 比较 al 与串 1 的字符 es:[edi]，并且 edi++。
        "jne 3f\n\t"                 // 如果不相等，则向前跳转到标号 3。
        "testb %%al,%%al\n\t"        // 该字符是 NULL 字符吗？
        "jne 1b\n"                   // 不是，则向后跳转到标号 1，继续比较。
        "2:\txorl %%eax,%%eax\n\t"   // 是 NULL 字符，则 eax 清零（返回值）。
        "jmp 4f\n"                   // 向前跳转到标号 4，结束。
        "3:\tmovl $1,%%eax\n\t"      // eax 中置 1。
        "jl 4f\n\t"                  // 如果前面比较中串 2 字符<串 2 字符，则返回 1，结束。
        "negl %%eax\n"               // 否则 eax = -eax，返回负值，结束。
        "4:"
        :"=a" (__res):"D" (cs),"S" (ct),"c" (count):"si","di","cx");
return __res;                       // 返回比较结果。
}
```

5.5 system_call.s 程序

5.5.1 功能描述

本程序主要实现系统调用(system_call)中断 int 0x80 的入口处理过程以及信号检测处理（从代码第 80 行开始），同时给出了两个系统功能的底层接口，分别是 sys_execve 和 sys_fork。还列出了处理过程类似的协处理器出错(int 16)、设备不存在(int7)、时钟中断(int32)、硬盘中断(int46)、软盘中断(int38)的中断处理程序。

对于软中断(system_call、coprocessor_error、device_not_available)，处理过程基本上是首先为调用相应 C 函数处理程序作准备，将一些参数压入堆栈，然后调用 C 函数进行相应功能的处理，处理返回后再去检测当前任务的信号位图，对值最小的一个信号进行处理并复位信号位图中的该信号。系统调用的 C 语言处理函数分布在整个 linux 内核代码中，由 include/linux/sys.h 头文件中的系统函数指针数组表来匹配。

对于硬件中断请求信号 IRQ 发来的中断，其处理过程首先是向中断控制芯片 8259A 发送结束硬件中断控制字指令 EOI，然后调用相应的 C 函数处理程序。对于时钟中断也要对当前任务的信号位图进行检测处理。

5.5.2 代码注释

列表 5.5 linux/kernel/system_call.s 程序

```
1 /*
2  *  linux/kernel/system_call.s
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7 /*
8  *  system_call.s  contains the system-call low-level handling routines.
9  * This also contains the timer-interrupt handler, as some of the code is
10 * the same. The hd- and flopppy-interrupts are also here.
11 *
12 * NOTE: This code handles signal-recognition, which happens every time
13 * after a timer-interrupt and after each system call. Ordinary interrupts
14 * don't handle signal-recognition, as that would clutter them up totally
15 * unnecessarily.
16 *
17 * Stack layout in 'ret_from_system_call':
18 *
19 *       0(%esp) - %eax
20 *       4(%esp) - %ebx
21 *       8(%esp) - %ecx
22 *       C(%esp) - %edx
23 *      10(%esp) - %fs
24 *      14(%esp) - %es
25 *      18(%esp) - %ds
26 *      1C(%esp) - %eip
27 *      20(%esp) - %cs
28 *      24(%esp) - %eflags
29 *      28(%esp) - %oldesp
30 *      2C(%esp) - %oldss
31 */
   /*
    * system_call.s 文件包含系统调用(system-call)底层处理子程序。由于有些代码比较类似，所以
    * 同时也包括时钟中断处理(timer-interrupt)句柄。硬盘和软盘的中断处理程序也在这里。
    *
    * 注意：这段代码处理信号(signal)识别，在每次时钟中断和系统调用之后都会进行识别。一般
    * 中断信号并不处理信号识别，因为会给系统造成混乱。
    *
    * 从系统调用返回（'ret_from_system_call'）时堆栈的内容见上面 19-30 行。
    */
32
33 SIG_CHLD        = 17          # 定义 SIG_CHLD 信号（子进程停止或结束）。
34
35 EAX             = 0x00        # 堆栈中各个寄存器的偏移位置。
36 EBX             = 0x04
37 ECX             = 0x08
38 EDX             = 0x0C
39 FS              = 0x10
40 ES              = 0x14
```

```
41 DS               = 0x18
42 EIP              = 0x1C
43 CS               = 0x20
44 EFLAGS           = 0x24
45 OLDESP           = 0x28       # 当有特权级变化时。
46 OLDSS            = 0x2C
47
   # 以下这些是任务结构(task_struct)中变量的偏移值，参见 include/linux/sched.h，77 行开始。
48 state   = 0              # these are offsets into the task-struct.  # 进程状态码
49 counter = 4              # 任务运行时间计数(递减)（滴答数），运行时间片。
50 priority = 8             // 运行优先数。任务开始运行时 counter=priority，越大则运行时间越长。
51 signal  = 12             // 是信号位图，每个比特位代表一种信号，信号值=位偏移值+1。
52 sigaction = 16           # MUST be 16 (=len of sigaction) // sigaction 结构长度必须是 16 字节。
                            // 信号执行属性结构数组的偏移值，对应信号将要执行的操作和标志信息。
53 blocked = (33*16)        // 受阻塞信号位图的偏移量。
54
   # 以下定义在 sigaction 结构中的偏移量，参见 include/signal.h，第 48 行开始。
55 # offsets within sigaction
56 sa_handler = 0           // 信号处理过程的句柄（描述符）。
57 sa_mask = 4              // 信号量屏蔽码
58 sa_flags = 8             // 信号集。
59 sa_restorer = 12         // 返回恢复执行的地址位置。
60
61 nr_system_calls = 72   # Linux 0.11 版内核中的系统调用总数。
62
63 /*
64  * Ok, I get parallel printer interrupts while using the floppy for some
65  * strange reason. Urgel. Now I just ignore them.
66  */
   /*
    * 好了，在使用软驱时我收到了并行打印机中断，很奇怪。呵，现在不管它。
    */
    # 定义入口点。
67 .globl _system_call,_sys_fork,_timer_interrupt,_sys_execve
68 .globl _hd_interrupt,_floppy_interrupt,_parallel_interrupt
69 .globl _device_not_available, _coprocessor_error
70
   # 错误的系统调用号。
71 .align 2       # 内存 4 字节对齐。
72 bad_sys_call:
73         movl $-1,%eax           # eax 中置-1，退出中断。
74         iret
   # 重新执行调度程序入口。调度程序 schedule 在(kernel/sched.c,104)。
75 .align 2
76 reschedule:
77         pushl $ret_from_sys_call  # 将 ret_from_sys_call 的地址入栈（101 行）。
78         jmp _schedule
   #### int 0x80 --linux 系统调用入口点(调用中断 int 0x80，eax 中是调用号)。
79 .align 2
80 _system_call:
81         cmpl $nr_system_calls-1,%eax    # 调用号如果超出范围的话就在 eax 中置-1 并退出。
82         ja bad_sys_call
83         push %ds                 # 保存原段寄存器值。
```

```
84          push %es
85          push %fs
86          pushl %edx              # ebx,ecx,edx 中放着系统调用相应的 C 语言函数的调用参数。
87          pushl %ecx              # push %ebx,%ecx,%edx as parameters
88          pushl %ebx              # to the system call
89          movl $0x10,%edx         # set up ds,es to kernel space
90          mov %dx,%ds             # ds,es 指向内核数据段(全局描述符表中数据段描述符)。
91          mov %dx,%es
92          movl $0x17,%edx         # fs points to local data space
93          mov %dx,%fs             # fs 指向局部数据段(局部描述符表中数据段描述符)。
    # 下面这句操作数的含义是：调用地址 = _sys_call_table + %eax * 4。参见列表后的说明。
    # 对应的 C 程序中的 sys_call_table 在 include/linux/sys.h 中，其中定义了一个包括 72 个
    # 系统调用 C 处理函数的地址数组表。
94          call _sys_call_table(,%eax,4)
95          pushl %eax              # 把系统调用号入栈。
96          movl _current,%eax      # 取当前任务（进程）数据结构地址➔eax。
    # 下面 97-100 行查看当前任务的运行状态。如果不在就绪状态(state 不等于 0)就去执行调度程序。
    # 如果该任务在就绪状态但 counter[??]值等于 0，则也去执行调度程序。
97          cmpl $0,state(%eax)          # state
98          jne reschedule
99          cmpl $0,counter(%eax)        # counter
100         je reschedule
    # 以下这段代码执行从系统调用 C 函数返回后，对信号量进行识别处理。
101 ret_from_sys_call:
    # 首先判别当前任务是否是初始任务 task0，如果是则不必对其进行信号量方面的处理，直接返回。
    # 103 行上的_task 对应 C 程序中的 task[]数组，直接引用 task 相当于引用 task[0]。
102         movl _current,%eax           # task[0] cannot have signals
103         cmpl _task,%eax
104         je 3f                        # 向前(forward)跳转到标号 3。
    # 通过对原调用程序代码选择符的检查来判断调用程序是否是超级用户。如果是超级用户就直接
    # 退出中断，否则需进行信号量的处理。这里比较选择符是否为普通用户代码段的选择符 0x000f
    # (RPL=3，局部表，第 1 个段(代码段))，如果不是则跳转退出中断程序。
105         cmpw $0x0f,CS(%esp)          # was old code segment supervisor ?
106         jne 3f
    # 如果原堆栈段选择符不为 0x17（也即原堆栈不在用户数据段中），则也退出。
107         cmpw $0x17,OLDSS(%esp)       # was stack segment = 0x17 ?
108         jne 3f
    # 下面这段代码（109-120）的用途是首先取当前任务结构中的信号位图(32 位，每位代表 1 种信号)，
    # 然后用任务结构中的信号阻塞（屏蔽）码，阻塞不允许的信号位，取得数值最小的信号值，再把
    # 原信号位图中该信号对应的位复位（置 0），最后将该信号值作为参数之一调用 do_signal()。
    # do_signal()在（kernel/signal.c,82）中，其参数包括 13 个入栈的信息。
109         movl signal(%eax),%ebx       # 取信号位图➔ebx，每 1 位代表 1 种信号，共 32 个信号。
110         movl blocked(%eax),%ecx      # 取阻塞（屏蔽）信号位图➔ecx。
111         notl %ecx                    # 每位取反。
112         andl %ebx,%ecx               # 获得许可的信号位图。
113         bsfl %ecx,%ecx               # 从低位（位 0）开始扫描位图，看是否有 1 的位，
                                         # 若有，则 ecx 保留该位的偏移值（即第几位 0-31）。
114         je 3f                        # 如果没有信号则向前跳转退出。
115         btrl %ecx,%ebx               # 复位该信号（ebx 含有原 signal 位图）。
116         movl %ebx,signal(%eax)       # 重新保存 signal 位图信息➔current->signal。
117         incl %ecx                    # 将信号调整为从 1 开始的数(1-32)。
118         pushl %ecx                   # 信号值入栈作为调用 do_signal 的参数之一。
119         call _do_signal              # 调用 C 函数信号处理程序(kernel/signal.c,82)
```

```
120         popl %eax                   # 弹出信号值。
121 3:      popl %eax
122         popl %ebx
123         popl %ecx
124         popl %edx
125         pop %fs
126         pop %es
127         pop %ds
128         iret
129
    #### int16 -- 下面这段代码处理协处理器发出的出错信号。跳转执行 C 函数 math_error()
    # (kernel/math/math_emulate.c,82)，返回后将跳转到 ret_from_sys_call 处继续执行。
130 .align 2
131 _coprocessor_error:
132         push %ds
133         push %es
134         push %fs
135         pushl %edx
136         pushl %ecx
137         pushl %ebx
138         pushl %eax
139         movl $0x10,%eax           # ds,es 置为指向内核数据段。
140         mov %ax,%ds
141         mov %ax,%es
142         movl $0x17,%eax           # fs 置为指向局部数据段（出错程序的数据段）。
143         mov %ax,%fs
144         pushl $ret_from_sys_call  # 把下面调用返回的地址入栈。
145         jmp _math_error           # 执行 C 函数 math_error()(kernel/math/math_emulate.c,37)
146
    #### int7 -- 设备不存在或协处理器不存在(Coprocessor not available)。
    # 如果控制寄存器 CR0 的 EM 标志置位，则当 CPU 执行一个 ESC 转义指令时就会引发该中断，这样就
    # 可以有机会让这个中断处理程序模拟 ESC 转义指令（169 行）。
    # CR0 的 TS 标志是在 CPU 执行任务转换时设置的。TS 可以用来确定什么时候协处理器中的内容（上下文）
    # 与 CPU 正在执行的任务不匹配了。当 CPU 在运行一个转义指令时发现 TS 置位了，就会引发该中断。
    # 此时就应该恢复新任务的协处理器执行状态（165 行）。参见(kernel/sched.c,77)中的说明。
    # 该中断最后将转移到标号 ret_from_sys_call 处执行下去（检测并处理信号）。
147 .align 2
148 _device_not_available:
149         push %ds
150         push %es
151         push %fs
152         pushl %edx
153         pushl %ecx
154         pushl %ebx
155         pushl %eax
156         movl $0x10,%eax           # ds,es 置为指向内核数据段。
157         mov %ax,%ds
158         mov %ax,%es
159         movl $0x17,%eax           # fs 置为指向局部数据段（出错程序的数据段）。
160         mov %ax,%fs
161         pushl $ret_from_sys_call  # 把下面跳转或调用的返回地址入栈。
162         clts                            # clear TS so that we can use math
163         movl %cr0,%eax
```

```
164         testl $0x4,%eax                  # EM (math emulation bit)
                                      # 如果不是 EM 引起的中断，则恢复新任务协处理器状态，
165         je _math_state_restore    # 执行 C 函数 math_state_restore()(kernel/sched.c,77)。
166         pushl %ebp
167         pushl %esi
168         pushl %edi
169         call _math_emulate        # 调用 C 函数 math_emulate(kernel/math/math_emulate.c,18)。
170         popl %edi
171         popl %esi
172         popl %ebp
173         ret                       # 这里的 ret 将跳转到 ret_from_sys_call(101 行)。
174
    #### int32 -- (int 0x20) 时钟中断处理程序。中断频率被设置为 100Hz(include/linux/sched.h,5),
    # 定时芯片 8253/8254 是在(kernel/sched.c,406)处初始化的。因此这里 jiffies 每 10 毫秒加 1。
    # 这段代码将 jiffies 增 1，发送结束中断指令给 8259 控制器，然后用当前特权级作为参数调用
    # C 函数 do_timer(long CPL)。当调用返回时转去检测并处理信号。
175 .align 2
176 _timer_interrupt:
177         push %ds                 # save ds,es and put kernel data space
178         push %es                 # into them. %fs is used by _system_call
179         push %fs
180         pushl %edx               # we save %eax,%ecx,%edx as gcc doesn't
181         pushl %ecx               # save those across function calls. %ebx
182         pushl %ebx               # is saved as we use that in ret_sys_call
183         pushl %eax
184         movl $0x10,%eax          # ds,es 置为指向内核数据段。
185         mov %ax,%ds
186         mov %ax,%es
187         movl $0x17,%eax          # fs 置为指向局部数据段（出错程序的数据段）。
188         mov %ax,%fs
189         incl _jiffies
    # 由于初始化中断控制芯片时没有采用自动 EOI，所以这里需要发指令结束该硬件中断。
190         movb $0x20,%al           # EOI to interrupt controller #1
191         outb %al,$0x20           # 操作命令字 OCW2 送 0x20 端口。
    # 下面 3 句从选择符中取出当前特权级别(0 或 3)并压入堆栈，作为 do_timer 的参数。
192         movl CS(%esp),%eax
193         andl $3,%eax             # %eax is CPL (0 or 3, 0=supervisor)
194         pushl %eax
195         call _do_timer           # 'do_timer(long CPL)' does everything from
196         addl $4,%esp             # task switching to accounting ...
197         jmp ret_from_sys_call
198
    #### 这是 sys_execve()系统调用。取中断调用程序的代码指针作为参数调用 C 函数 do_execve()。
    # do_execve()在(fs/exec.c,182)。
199 .align 2
200 _sys_execve:
201         lea EIP(%esp),%eax
202         pushl %eax
203         call _do_execve
204         addl $4,%esp             # 丢弃调用时压入栈的 EIP 值。
205         ret
206
    #### sys_fork()调用，用于创建子进程，是 system_call 功能 2。原形在 include/linux/sys.h 中。
```

```
# 首先调用 C 函数 find_empty_process()，取得一个进程号 pid。若返回负数则说明目前任务数组
# 已满。然后调用 copy_process()复制进程。
.align 2
_sys_fork:
        call _find_empty_process    # 调用 find_empty_process()(kernel/fork.c,135)。
        testl %eax,%eax
        js 1f
        push %gs
        pushl %esi
        pushl %edi
        pushl %ebp
        pushl %eax
        call _copy_process          # 调用 C 函数 copy_process()(kernel/fork.c,68)。
        addl $20,%esp               # 丢弃这里所有压栈内容。
1:      ret

#### int 46 -- (int 0x2E) 硬盘中断处理程序，响应硬件中断请求 IRQ14。
# 当硬盘操作完成或出错就会发出此中断信号。(参见 kernel/blk_drv/hd.c)。
# 首先向 8259A 中断控制从芯片发送结束硬件中断指令(EOI)，然后取变量 do_hd 中的函数指针放入 edx
# 寄存器中，并置 do_hd 为 NULL，接着判断 edx 函数指针是否为空。如果为空，则给 edx 赋值指向
# unexpected_hd_interrupt()，用于显示出错信息。随后向 8259A 主芯片送 EOI 指令，并调用 edx 中
# 指针指向的函数: read_intr()、write_intr()或 unexpected_hd_interrupt()。
_hd_interrupt:
        pushl %eax
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%eax           # ds,es 置为内核数据段。
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax           # fs 置为调用程序的局部数据段。
        mov %ax,%fs
# 由于初始化中断控制芯片时没有采用自动 EOI，所以这里需要发指令结束该硬件中断。
        movb $0x20,%al
        outb %al,$0xA0            # EOI to interrupt controller #1    # 送从 8259A。
        jmp 1f                    # give port chance to breathe
1:      jmp 1f                    # 延时作用。
1:      xorl %edx,%edx
        xchgl _do_hd,%edx         # do_hd 定义为一个函数指针，将被赋值 read_intr()或
                                  # write_intr()函数地址。(kernel/blk_drv/hd.c)
                                  # 放到 edx 寄存器后就将 do_hd 指针变量置为 NULL。
        testl %edx,%edx           # 测试函数指针是否为 Null。
        jne 1f                    # 若空，则使指针指向 C 函数 unexpected_hd_interrupt()。
        movl $_unexpected_hd_interrupt,%edx    # (kernel/blk_drv/hdc,237)。
1:      outb %al,$0x20            # 送主 8259A 中断控制器 EOI 指令（结束硬件中断）。
        call *%edx                # "interesting" way of handling intr.
        pop %fs                   # 上句调用 do_hd 指向的 C 函数。
        pop %es
        pop %ds
        popl %edx
        popl %ecx
```

```
        popl %eax
        iret

#### int38 -- (int 0x26) 软盘驱动器中断处理程序，响应硬件中断请求 IRQ6。
# 其处理过程与上面对硬盘的处理基本一样。(kernel/blk_drv/floppy.c)。
# 首先向 8259A 中断控制器主芯片发送 EOI 指令，然后取变量 do_floppy 中的函数指针放入 eax
# 寄存器中，并置 do_floppy 为 NULL，接着判断 eax 函数指针是否为空。如为空，则给 eax 赋值指向
# unexpected_floppy_interrupt ()，用于显示出错信息。随后调用 eax 指向的函数: rw_interrupt,
# seek_interrupt,recal_interrupt,reset_interrupt 或 unexpected_floppy_interrupt。
_floppy_interrupt:
        pushl %eax
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%eax         # ds,es 置为内核数据段。
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax         # fs 置为调用程序的局部数据段。
        mov %ax,%fs
        movb $0x20,%al          # 送主 8259A 中断控制器 EOI 指令（结束硬件中断）。
        outb %al,$0x20          # EOI to interrupt controller #1
        xorl %eax,%eax
        xchgl _do_floppy,%eax   # do_floppy 为一函数指针，将被赋值实际处理 C 函数程序，
                                # 放到 eax 寄存器后就将 do_floppy 指针变量置空。
        testl %eax,%eax         # 测试函数指针是否=NULL?
        jne 1f                  # 若空，则使指针指向 C 函数 unexpected_floppy_interrupt()。
        movl $_unexpected_floppy_interrupt,%eax
1:      call *%eax              # "interesting" way of handling intr.
        pop %fs                 # 上句调用 do_floppy 指向的函数。
        pop %es
        pop %ds
        popl %edx
        popl %ecx
        popl %eax
        iret

#### int 39 -- (int 0x27) 并行口中断处理程序，对应硬件中断请求信号 IRQ7。
# 本版本内核还未实现。这里只是发送 EOI 指令。
_parallel_interrupt:
        pushl %eax
        movb $0x20,%al
        outb %al,$0x20
        popl %eax
        iret
```

5.5.3 其它信息

5.5.3.1 GNU 汇编语言的 32 位寻址方式

采用的是 AT&T 的汇编语言语法。32 位寻址的正规格式为：

AT&T: immed32(basepointer, indexpointer, indexscale)
Intel: [basepointer + indexpointer*indexscal + immed32]

该格式寻址位置的计算方式为：immed32 + basepointer + indexpointer * indexscale

在应用时，并不需要写出所有这些字段，但 immed32 和 basepointer 之中必须有一个存在。以下是一些例子。

o 对一个指定的 C 语言变量寻址：

AT&T: _booga　　　　　　　　　Intel: [_booga]

注意：变量前的下划线是从汇编程序中得到静态（全局）C 变量(booga)的方法。

o 对寄存器内容指向的位置寻址：

AT&T: (%eax)　　　　　　　　　Intel: [eax]

o 通过寄存器中的内容作为基址寻址一个变量：

AT&T: _variable(%eax)　　　　　　Intel: [eax + _variable]

o 在一个整数数组中寻址一个值（比例值为 4）：

AT&T: _array(,%eax,4)　　　　　　Intel: [eax*4 + _array]

o 使用直接数寻址偏移量：

对于 C 语言：*(p+1) 其中 p 是字符的指针 char *

AT&T: 则 AT&T 格式：1(%eax) 其中 eax 中是 p 的值。　　Intel: [eax+1]

o 在一个 8 字节为一个记录的数组中寻址指定的字符。其中 eax 中是指定的记录号，ebx 中是指定字符在记录中的偏移址：

AT&T: _array(%ebx,%eax,8)　　　　Intel: [ebx + eax*8 + _array]

5.6 mktime.c 文件

5.6.1 功能描述

该该程序只有一个函数 mktime()，仅供内核使用。计算从 1970 年 1 月 1 日 0 时起到开机当日经过的秒数，作为开机时间。

5.6.2 代码注释

列表 5.6 linux/kernel/mktime.c 程序

```
/*
 *  linux/kernel/mktime.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <time.h>              // 时间头文件，定义了标准时间数据结构 tm 和一些处理时间函数原型。

/*
 * This isn't the library routine, it is only used in the kernel.
 * as such, we don't care about years<1970 etc, but assume everything
 * is ok. Similarly, TZ etc is happily ignored. We just do everything
```

```
 * as easily as possible. Let's find something public for the library
 * routines (although I think minix times is public).
 */
/*
 * PS. I hate whoever though up the year 1970 - couldn't they have gotten
 * a leap-year instead? I also hate Gregorius, pope or no. I'm grumpy.
 */
/*
 * 这不是库函数，它仅供内核使用。因此我们不关心小于 1970 年的年份等，但假定一切均很正常。
 * 同样，时间区域 TZ 问题也先忽略。我们只是尽可能简单地处理问题。最好能找到一些公开的库函数
 * （尽管我认为 minix 的时间函数是公开的）。
 * 另外，我恨那个设置 1970 年开始的人 - 难道他们就不能选择从一个闰年开始？我也恨罗马教皇、
 * 主教或者什么也不在乎。我是个脾气暴躁的人。
 */
#define MINUTE 60                      // 1 分钟的秒数。
#define HOUR (60*MINUTE)               // 1 小时的秒数。
#define DAY (24*HOUR)                  // 1 天的秒数。
#define YEAR (365*DAY)                 // 1 年的秒数。

/* interestingly, we assume leap-years */
/* 有趣的是我们考虑进了闰年 */
// 下面以年为界限，定义了每个月开始时的秒数时间数组。
static int month[12] = {
        0,
        DAY*(31),
        DAY*(31+29),
        DAY*(31+29+31),
        DAY*(31+29+31+30),
        DAY*(31+29+31+30+31),
        DAY*(31+29+31+30+31+30),
        DAY*(31+29+31+30+31+30+31),
        DAY*(31+29+31+30+31+30+31+31),
        DAY*(31+29+31+30+31+30+31+31+30),
        DAY*(31+29+31+30+31+30+31+31+30+31),
        DAY*(31+29+31+30+31+30+31+31+30+31+30)
};

// 该函数计算从 1970 年 1 月 1 日 0 时起到开机当日经过的秒数，作为开机时间。
long kernel_mktime(struct tm * tm)
{
        long res;
        int year;

        year = tm->tm_year - 70;           // 从 70 年到现在经过的年数(2 位表示方式)，
                                           // 因此会有 2000 年问题。
/* magic offsets (y+1) needed to get leapyears right.*/
/* 为了获得正确的闰年数，这里需要这样一个魔幻偏值(y+1) */
        res = YEAR*year + DAY*((year+1)/4); // 这些年经过的秒数时间 + 每个闰年时多 1 天
        res += month[tm->tm_mon];          // 的秒数时间，在加上当年到当月时的秒数。
/* and (y+2) here. If it wasn't a leap-year, we have to adjust */
/* 以及(y+2)。如果(y+2)不是闰年，那么我们就必须进行调整(减去一天的秒数时间)。*/
        if (tm->tm_mon>1 && ((year+2)%4))
                res -= DAY;
```

```
        res += DAY*(tm->tm_mday-1);          // 再加上本月过去的天数的秒数时间。
        res += HOUR*tm->tm_hour;             // 再加上当天过去的小时数的秒数时间。
        res += MINUTE*tm->tm_min;            // 再加上 1 小时内过去的分钟数的秒数时间。
        res += tm->tm_sec;                   // 再加上 1 分钟内已过的秒数。
        return res;                          // 即等于从 1970 年以来经过的秒数时间。
}

```

5.6.3 其它信息

5.6.3.1 闰年时间的计算方法

闰年的基本计算方法是：

如果 y 能被 4 除尽且不能被 100 除尽，或者能被 400 除尽，则 y 是闰年。

5.7 sched.c 文件

5.7.1 功能描述

sched.c 是内核中有关任务调度函数的程序，其中包括有关调度的基本函数(sleep_on、wakeup、schedule 等)以及一些简单的系统调用函数（比如 getpid()）。另外 Linus 为了编程的方便，考虑到软盘驱动器程序定时的需要，也将操作软盘的几个函数放到了这里。

这几个基本函数的代码虽然不长，但有些抽象，比较难以理解。好在世面上有很多教科书对此解释得都很清楚，因此可以参考其它书籍对这些函数的讨论。这些也就是教科书上的重点讲述对象，否则理论书籍也就没有什么好讲的了☺。

首先参考包含文件 include/kernel/sched.h 来阅读代码，然后再看代码后对几个重要函数的说明，以便更清晰地了解内核的调度机理。

5.7.2 代码注释

列表 5.7 linux/kernel/sched.c 程序

```
/*
 *  linux/kernel/sched.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 'sched.c' is the main kernel file. It contains scheduling primitives
 * (sleep_on, wakeup, schedule etc) as well as a number of simple system
 * call functions (type getpid(), which just extracts a field from
 * current-task
 */
/*
 * 'sched.c'是主要的内核文件。其中包括有关调度的基本函数(sleep_on、wakeup、schedule 等)以及
 * 一些简单的系统调用函数（比如 getpid()，仅从当前任务中获取一个字段）。
 */
#include <linux/sched.h>  // 调度程序头文件。定义了任务结构 task_struct、第 1 个初始任务
                          // 的数据。还有一些以宏的形式定义的有关描述符参数设置和获取的
```

```
                               // 嵌入式汇编函数程序。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <linux/sys.h>    // 系统调用头文件。含有 72 个系统调用 C 函数处理程序,以'sys_'开头。
#include <linux/fdreg.h>  // 软驱头文件。含有软盘控制器参数的一些定义。
#include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
#include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
#include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

#include <signal.h>       // 信号头文件。定义信号符号常量，sigaction 结构，操作函数原型。

#define _S(nr) (1<<((nr)-1)) // 取信号 nr 在信号位图中对应位的二进制数值。信号编号 1-32。
                             // 比如信号 5 的位图数值 = 1<<(5-1) = 16 = 00010000b。
#define _BLOCKABLE (~(_S(SIGKILL) | _S(SIGSTOP))) // 除了 SIGKILL 和 SIGSTOP 信号以外其它都是
                                                  // 可阻塞的(…10111111111011111111b)。

// 显示任务号 nr 的进程号、进程状态和内核堆栈空闲字节数（大约）。
void show_task(int nr,struct task_struct * p)
{
        int i,j = 4096-sizeof(struct task_struct);

        printk("%d: pid=%d, state=%d, ",nr,p->pid,p->state);
        i=0;
        while (i<j && !((char *)(p+1))[i])  // 检测指定任务数据结构以后等于 0 的字节数。
                i++;
        printk("%d (of %d) chars free in kernel stack\n\r",i,j);
}

// 显示所有任务的任务号、进程号、进程状态和内核堆栈空闲字节数（大约）。
void show_stat(void)
{
        int i;

        for (i=0;i<NR_TASKS;i++)    // NR_TASKS 是系统能容纳的最大进程（任务）数量（64 个），
                if (task[i])        // 定义在 include/kernel/sched.h 第 4 行。
                        show_task(i,task[i]);
}

// 定义每个时间片的滴答数☺。
#define LATCH (1193180/HZ)

extern void mem_use(void);          // [??]没有任何地方定义和引用该函数。

extern int timer_interrupt(void);   // 时钟中断处理程序(kernel/system_call.s,176)。
extern int system_call(void);       // 系统调用中断处理程序(kernel/system_call.s,80)。

union task_union {                  // 定义任务联合(任务结构成员和 stack 字符数组程序成员)。
        struct task_struct task;    // 因为一个任务数据结构与其堆栈放在同一内存页中，所以
        char stack[PAGE_SIZE];      // 从堆栈段寄存器 ss 可以获得其数据段选择符。
};

static union task_union init_task = {INIT_TASK,};  // 定义初始任务的数据(sched.h 中)。

long volatile jiffies=0;            // 从开机开始算起的滴答数时间值（10ms/滴答）。
```

```
// 前面的限定符 volatile，英文解释是易变、不稳定的意思。这里是要求 gcc 不要对该变量进行优化
// 处理，也不要挪动位置，因为也许别的程序会来修改它的值。
long startup_time=0;                         // 开机时间。从 1970:0:0:0 开始计时的秒数。
struct task_struct *current = &(init_task.task); // 当前任务指针（初始化为初始任务）。
struct task_struct *last_task_used_math = NULL;  // 使用过协处理器任务的指针。

struct task_struct * task[NR_TASKS] = {&(init_task.task), }; // 定义任务指针数组。

long user_stack [ PAGE_SIZE>>2 ] ;  // 定义用户堆栈数组（1024 项）。

struct {                            // 堆栈开始结构（地址指针，数据段选择符）。
        long * a;
        short b;
        } stack_start = { & user_stack [PAGE_SIZE>>2] , 0x10 };
/*
 *  'math_state_restore()' saves the current math information in the
 * old math state array, and gets the new ones from the current task
 */
/*
 * 将当前协处理器内容保存到老协处理器状态数组中，并将当前任务的协处理器
 * 内容加载进协处理器。
 */
// 当任务被调度交换过以后，该函数用以保存原任务的协处理器状态（上下文）并恢复新调度进来的
// 当前任务的协处理器执行状态。
void math_state_restore()
{
        if (last_task_used_math == current) // 如果任务没变则返回(上一个任务就是当前任务)。
                return;                     // 这里所指的"上一个任务"是刚被交换出去的任务。
        __asm__("fwait");                   // 在发送协处理器命令之前要先发 WAIT 指令。
        if (last_task_used_math) {          // 如果上个任务使用了协处理器，则保存其状态。
                __asm__("fnsave %0"::"m" (last_task_used_math->tss.i387));
        }
        last_task_used_math=current;        // 现在，last_task_used_math 指向当前任务，
                                            // 以备当前任务被交换出去时使用。
        if (current->used_math) {           // 如果当前任务用过协处理器，则恢复其状态。
                __asm__("frstor %0"::"m" (current->tss.i387));
        } else {                            // 否则的话说明是第一次使用，
                __asm__("fninit"::);        // 于是就向协处理器发初始化命令，
                current->used_math=1;       // 并设置使用了协处理器标志。
        }
}

/*
 *  'schedule()' is the scheduler function. This is GOOD CODE! There
 * probably won't be any reason to change this, as it should work well
 * in all circumstances (ie gives IO-bound processes good response etc).
 * The one thing you might take a look at is the signal-handler code here.
 *
 *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */
/*
```

```
 * 'schedule()'是调度函数。这是个很好的代码！没有任何理由对它进行修改，因为它可以在所有的
 * 环境下工作（比如能够对 IO-边界处理很好的响应等）。只有一件事值得留意，那就是这里的信号
 * 处理代码。
 *   注意！！任务 0 是个闲置('idle')任务，只有当没有其它任务可以运行时才调用它。它不能被杀
 * 死，也不能睡眠。任务 0 中的状态信息'state'是从来不用的。
 */
104 void schedule(void)
105 {
106         int i,next,c;
107         struct task_struct ** p;            // 任务结构指针的指针。
108
109 /* check alarm, wake up any interruptible tasks that have got a signal */
    /* 检测 alarm（进程的报警定时值），唤醒任何得到信号的可中断任务 */
110
    // 从任务数组中最后一个任务开始检测 alarm。
111         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
112                 if (*p) {
    // 如果任务的 alarm 时间已经过期(alarm<jiffies),则在信号位图中置 SIGALRM 信号，然后清 alarm。
    // jiffies 是系统从开机开始算起的滴答数（10ms/滴答）。定义在 sched.h 第 139 行。
113                         if ((*p)->alarm && (*p)->alarm < jiffies) {
114                                         (*p)->signal |= (1<<(SIGALRM-1));
115                                         (*p)->alarm = 0;
116                                 }
    // 如果信号位图中除去被阻塞的信号外还有其它信号并且任务处于可中断状态，则置任务为就绪状态。
    // 其中'~(_BLOCKABLE & (*p)->blocked)'用于忽略被阻塞的信号，但 SIGKILL 和 SIGSTOP 不能被阻塞。
117                         if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
118                         (*p)->state==TASK_INTERRUPTIBLE)
119                                 (*p)->state=TASK_RUNNING;       //置为就绪（可执行）状态。
120                 }
121
122 /* this is the scheduler proper: */
    /* 这里是调度程序的主要部分 */
123
124         while (1) {
125                 c = -1;
126                 next = 0;
127                 i = NR_TASKS;
128                 p = &task[NR_TASKS];
    // 这段代码也是从任务数组的最后一个任务开始循环处理，并跳过不含任务的任务槽。比较每个就绪
    // 状态任务的 counter（任务运行时间的递减滴答计数）值，哪一个值大，运行时间还不长，next 就
    // 指向哪个的任务号。
129                 while (--i) {
130                         if (!*--p)
131                                 continue;
132                         if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
133                                 c = (*p)->counter, next = i;
134                 }
    // 如果比较得出有 counter 值大于 0 的结果，则退出 124 行开始的循环，执行任务切换（141 行）。
135                 if (c) break;
    // 否则就根据每个任务的优先权值，更新每一个任务的 counter 值，然后回到 125 行重新比较。
    // counter 值的计算方式为 counter = counter /2 + priority。[右边 counter=0??]
136                 for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
137                         if (*p)
```

```
                                (*p)->counter = ((*p)->counter >> 1) +
                                                (*p)->priority;
        }
        switch_to(next);    // 切换到任务号为 next 的任务，并运行之。
}

// pause()系统调用。转换当前任务的状态为可中断的等待状态，并重新调度。
int sys_pause(void)
{
        current->state = TASK_INTERRUPTIBLE;
        schedule();
        return 0;
}

// 把当前任务置为不可中断的等待状态，并让睡眠队列头的指针指向当前任务。
// 只有明确地唤醒时才会返回。该函数提供了进程与中断处理程序之间的同步机制。
// 函数参数*p 是放置等待任务的队列头指针。（参见列表后的说明）。
void sleep_on(struct task_struct **p)
{
        struct task_struct *tmp;

// 若指针无效，则退出。（指针所指的对象可以是 NULL，但指针本身不会为 0)。
        if (!p)
                return;
        if (current == &(init_task.task)) // 如果当前任务是任务 0，则死机(impossible!)。
                panic("task[0] trying to sleep");
        tmp = *p;                         // 让 tmp 指向已经在等待队列上的任务(如果有的话)。
        *p = current;                     // 将睡眠队列头的等待指针指向当前任务。
        current->state = TASK_UNINTERRUPTIBLE; // 将当前任务置为不可中断的等待状态。
        schedule();                       // 重新调度。
// 只有当这个等待任务被唤醒时，调度程序才又返回到这里，则表示被明确地唤醒。
        if (tmp)                          // 若还存在等待的任务，则也将其置为就绪状态。
                tmp->state=0;
}

// 将当前任务置为可中断的等待状态，并放入*p 指定的等待队列中。参见列表后对 sleep_on()的说明。
void interruptible_sleep_on(struct task_struct **p)
{
        struct task_struct *tmp;

        if (!p)
                return;
        if (current == &(init_task.task))
                panic("task[0] trying to sleep");
        tmp=*p;
        *p=current;
repeat: current->state = TASK_INTERRUPTIBLE;
        schedule();
// 如果等待队列中还有等待任务，并且队列头指针所指向的任务不是当前任务时，则将该等待任务置为
// 可运行的就绪状态，并重新执行调度程序。当指针*p 所指向的不是当前任务时，表示在当前任务被放
// 入队列后，又有新的任务被插入等待队列中，因此，既然本任务是可中断的，就应该首先执行所有
// 其它的等待任务。
        if (*p && *p != current) {
```

```
                (**p).state=0;
                goto repeat;
        }
// 下面一句代码有误，应该是*p = tmp，让队列头指针指向其余等待任务，否则在当前任务之前插入
// 等待队列的任务均被抹掉了。参见图 4.3。
        *p=NULL;
        if (tmp)
                tmp->state=0;
}

// 唤醒指定任务*p。
void wake_up(struct task_struct **p)
{
        if (p && *p) {
                (**p).state=0;   // 置为就绪（可运行）状态。
                *p=NULL;
        }
}

/*
 * OK, here are some floppy things that shouldn't be in the kernel
 * proper. They are here because the floppy needs a timer, and this
 * was the easiest way of doing it.
 */
/*
 * 好了，从这里开始是一些有关软盘的子程序，本不应该放在内核的主要部分中的。将它们放在这里
 * 是因为软驱需要一个时钟，而放在这里是最方便的办法。
 */
static struct task_struct * wait_motor[4] = {NULL,NULL,NULL,NULL};
static int  mon_timer[4]={0,0,0,0};
static int moff_timer[4]={0,0,0,0};
unsigned char current_DOR = 0x0C;   // 数字输出寄存器(初值：允许 DMA 和请求中断、启动 FDC)。

// 指定软盘到正常运转状态所需延迟滴答数（时间）。
// nr -- 软驱号(0-3)，返回值为滴答数。
int ticks_to_floppy_on(unsigned int nr)
{
        extern unsigned char selected;    // 当前选中的软盘号(kernel/blk_drv/floppy.c,122)。
        unsigned char mask = 0x10 << nr;  // 所选软驱对应数字输出寄存器中启动马达比特位。

        if (nr>3)
                panic("floppy_on: nr>3"); // 最多 4 个软驱。
        moff_timer[nr]=10000;           /* 100 s = very big :-) */
        cli();                          /* use floppy_off to turn it off */
        mask |= current_DOR;
// 如果不是当前软驱，则首先复位其它软驱的选择位，然后置对应软驱选择位。
        if (!selected) {
                mask &= 0xFC;
                mask |= nr;
        }
// 如果数字输出寄存器的当前值与要求的值不同，则向 FDC 数字输出端口输出新值(mask)。并且如果
// 要求启动的马达还没有启动，则置相应软驱的马达启动定时器值(HZ/2 = 0.5 秒或 50 个滴答)。
// 此后更新当前数字输出寄存器值 current_DOR。
```

```
        if (mask != current_DOR) {
                outb(mask,FD_DOR);
                if ((mask ^ current_DOR) & 0xf0)
                        mon_timer[nr] = HZ/2;
                else if (mon_timer[nr] < 2)
                        mon_timer[nr] = 2;
                current_DOR = mask;
        }
        sti();
        return mon_timer[nr];
}

// 等待指定软驱马达启动所需时间。
void floppy_on(unsigned int nr)
{
        cli();   // 关中断。
        while (ticks_to_floppy_on(nr))     // 如果马达启动定时还没到，就一直把当前进程置
                sleep_on(nr+wait_motor);  // 为不可中断睡眠状态并放入等待马达运行的队列中。
        sti();   // 开中断。
}

// 置关闭相应软驱马达停转定时器（3 秒）。
void floppy_off(unsigned int nr)
{
        moff_timer[nr]=3*HZ;
}

// 软盘定时处理子程序。更新马达启动定时值和马达关闭停转计时值。该子程序是在时钟定时
// 中断中被调用，因此每一个滴答(10ms)被调用一次，更新马达开启或停转定时器的值。如果某
// 一个马达停转定时到，则将数字输出寄存器马达启动位复位。
void do_floppy_timer(void)
{
        int i;
        unsigned char mask = 0x10;

        for (i=0 ; i<4 ; i++,mask <<= 1) {
                if (!(mask & current_DOR))             // 如果不是 DOR 指定的马达则跳过。
                        continue;
                if (mon_timer[i]) {
                        if (!--mon_timer[i])
                                wake_up(i+wait_motor); // 如果马达启动定时到则唤醒进程。
                } else if (!moff_timer[i]) {           // 如果马达停转定时到则
                        current_DOR &= ~mask;          // 复位相应马达启动位，并
                        outb(current_DOR,FD_DOR);      // 更新数字输出寄存器。
                } else
                        moff_timer[i]--;               // 马达停转计时递减。
        }
}

#define TIME_REQUESTS 64              // 最多可有 64 个定时器链表（64 个任务）。

// 定时器链表结构和定时器数组。
static struct timer_list {
```

```
        long jiffies;               // 定时滴答数。
        void (*fn)();               // 定时处理程序。
        struct timer_list * next;   // 下一个定时器。
} timer_list[TIME_REQUESTS], * next_timer = NULL;

// 添加定时器。输入参数为指定的定时值(滴答数)和相应的处理程序指针。
//
void add_timer(long jiffies, void (*fn)(void))
{
        struct timer_list * p;

// 如果定时处理程序指针为空，则退出。
        if (!fn)
                return;
        cli();
// 如果定时值<=0，则立刻调用其处理程序。并且该定时器不加入链表中。
        if (jiffies <= 0)
                (fn)();
        else {
// 从定时器数组中，找一个空闲项。
                for (p = timer_list ; p < timer_list + TIME_REQUESTS ; p++)
                        if (!p->fn)
                                break;
// 如果已经用完了定时器数组，则系统崩溃☺。
                if (p >= timer_list + TIME_REQUESTS)
                        panic("No more time requests free");
// 向定时器数据结构填入相应信息。并链入链表头
                p->fn = fn;
                p->jiffies = jiffies;
                p->next = next_timer;
                next_timer = p;
// 链表项按定时值从小到大排序。在排序时减去排在前面需要的滴答数，这样在处理定时器时只要
// 查看链表头的第一项的定时是否到期即可。[[?? 这段程序好象没有考虑周全。如果新插入的定时
// 器值<原来头一个定时器值时，也应该将所有后面的定时值均减去新的第 1 个的定时值。]]
                while (p->next && p->next->jiffies < p->jiffies) {
                        p->jiffies -= p->next->jiffies;
                        fn = p->fn;
                        p->fn = p->next->fn;
                        p->next->fn = fn;
                        jiffies = p->jiffies;
                        p->jiffies = p->next->jiffies;
                        p->next->jiffies = jiffies;
                        p = p->next;
                }
        }
        sti();
}

// 时钟中断 C 函数处理程序。
void do_timer(long cpl)
{
        extern int beepcount;   // 扬声器发声时间滴答数(kernel/chr_drv/console.c,697)
        extern void sysbeepstop(void);  // 关闭扬声器(kernel/chr_drv/console.c,691)
```

```
309
    // 如果发声计数次数到，则关闭发声。(向 0x61 口发送命令，复位位 0 和 1。位 0 控制 8253
    // 计数器 2 的工作，位 1 控制扬声器)。
310         if (beepcount)
311                 if (!--beepcount)
312                         sysbeepstop();
313
    // 如果当前特权级(cpl)为 0（最高，表示是内核程序在工作），则将超级用户运行时间 stime 递增；
    // 如果 cpl > 0，则表示是一般用户程序在工作，增加 utime。
314         if (cpl)
315                 current->utime++;
316         else
317                 current->stime++;
318
    // 如果有用户的定时器存在，则将链表第 1 个定时器的值减 1。如果已等于 0，则调用相应的处理
    // 程序，并将该处理程序指针置为空。然后去掉该项定时器。
319         if (next_timer) {                       // next_timer 是定时器链表的头指针(见 270 行)。
320                 next_timer->jiffies--;
321                 while (next_timer && next_timer->jiffies <= 0) {
322                         void (*fn)(void);    // 这里插入了一个函数指针定义！！！☹
323
324                         fn = next_timer->fn;
325                         next_timer->fn = NULL;
326                         next_timer = next_timer->next;
327                         (fn)();                 // 调用处理函数。
328                 }
329         }
    // 如果当前软盘控制器 FDC 的数字输出寄存器中马达启动位有置位的，则执行软盘定时程序(245 行)。
330         if (current_DOR & 0xf0)
331                 do_floppy_timer();
332         if ((--current->counter)>0) return;  // 如果进程运行时间还没完，则退出。
333         current->counter=0;
334         if (!cpl) return;   // 对于超级用户程序，不依赖 counter 值进行调度。
335         schedule();
336 }
337
    // 系统调用功能 - 设置报警定时时间值(秒)。
    // 如果已经设置过 alarm 值，则返回旧值，否则返回 0。
338 int sys_alarm(long seconds)
339 {
340         int old = current->alarm;
341
342         if (old)
343                 old = (old - jiffies) / HZ;
344         current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
345         return (old);
346 }
347
    // 取当前进程号 pid。
348 int sys_getpid(void)
349 {
350         return current->pid;
351 }
```

```
// 取父进程号 ppid。
int sys_getppid(void)
{
        return current->father;
}

// 取用户号 uid。
int sys_getuid(void)
{
        return current->uid;
}

// 取 euid。
int sys_geteuid(void)
{
        return current->euid;
}

// 取组号 gid。
int sys_getgid(void)
{
        return current->gid;
}

// 取 egid。
int sys_getegid(void)
{
        return current->egid;
}

// 系统调用功能 -- 降低对 CPU 的使用优先权（有人会用吗？☺）。
// 应该限制 increment 大于 0，否则的话,可使优先权增大！！
int sys_nice(long increment)
{
        if (current->priority-increment>0)
                current->priority -= increment;
        return 0;
}

// 调度程序的初始化子程序。
void sched_init(void)
{
        int i;
        struct desc_struct * p;   // 描述符表结构指针。

        if (sizeof(struct sigaction) != 16)  // sigaction 是存放有关信号状态的结构。
                panic("Struct sigaction MUST be 16 bytes");
// 设置初始任务（任务 0）的任务状态段描述符和局部数据表描述符(include/asm/system.h,65)。
        set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
        set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
// 清任务数组和描述符表项（注意 i=1 开始，所以初始任务的描述符还在）。
        p = gdt+2+FIRST_TSS_ENTRY;
```

```
        for(i=1;i<NR_TASKS;i++) {
                task[i] = NULL;
                p->a=p->b=0;
                p++;
                p->a=p->b=0;
                p++;
        }
/* Clear NT, so that we won't have troubles with that later on */
/* 清除标志寄存器中的位 NT，这样以后就不会有麻烦 */
// NT 标志用于控制程序的递归调用(Nested Task)。当 NT 置位时，那么当前中断任务执行
// iret 指令时就会引起任务切换。NT 指出 TSS 中的 back_link 字段是否有效。
        __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");  // 复位 NT 标志。
        ltr(0);                          // 将任务 0 的 TSS 加载到任务寄存器 tr。
        lldt(0);                         // 将局部描述符表加载到局部描述符表寄存器。
// 注意！！是将 GDT 中相应 LDT 描述符的选择符加载到 ldtr。只明确加载这一次，以后新任务
// LDT 的加载，是 TSS 根据其中的 LDT 项自动加载。

// 下面代码用于初始化 8253 定时器。
        outb_p(0x36,0x43);              /* binary, mode 3, LSB/MSB, ch 0 */
        outb_p(LATCH & 0xff , 0x40);    /* LSB */   // 定时值低字节。
        outb(LATCH >> 8 , 0x40);        /* MSB */   // 定时值高字节。
// 设置时钟中断处理程序句柄（设置时钟中断门）。
        set_intr_gate(0x20,&timer_interrupt);
// 修改中断控制器屏蔽码，允许时钟中断。
        outb(inb_p(0x21)&~0x01,0x21);
// 设置系统调用中断门。
        set_system_gate(0x80,&system_call);
}
```

5.7.3 其它信息

5.7.3.1 schedule()函数说明

schedule()函数首先检查任务的信号。对于每个任务的报警定时值，若定时值已到或超过，则在该任务的信号位图中置 SIGALRM 信号。如果任务的信号位图中除去被阻塞的信号外还有其它信号并且任务处于可中断状态，则置任务为就绪状态。

然后是调度程序的核心处理部分。首先循环检查任务数组中的所有任务，根据每个就绪态任务剩余执行时间的值 counter，选取该值最大的一个任务，并利用 switch_to()函数切换到该任务。若所有就绪态任务的该值都等于零，表示此刻所有任务的时间片都已经运行完，于是就根据任务的优先权值 priority，重置每个任务的运行时间片值 counter，再重新执行循环检查所有任务的执行时间片值。

5.7.3.2 sleep_on()函数说明

sleep_on()函数虽然很短，却要比 schedule()函数难理解。这里用图示的方法加以解释。简单地说，sleep_on()函数的主要功能是当一个进程（或任务）所请求的资源正忙或不在内存中时暂时切换出去，放在等待队列中等待一段时间。当切换回来后再继续运行。放入等待队列的方式是利用了函数中的 tmp 指针作为各个正在等待任务的联系。

函数中共牵涉到对三个任务指针操作：*p、tmp 和 current，*p 是等待队列头指针，如文件系统内存 i 节点的 i_wait 指针、内存缓冲操作中的 buffer_wait 指针等；tmp 是临时指针；current 是当前任务指针。对于这些指针在内存中的变化情况我们可以用下面的示意图说明（图 5.4）。图中的长条表示内存字节序列。

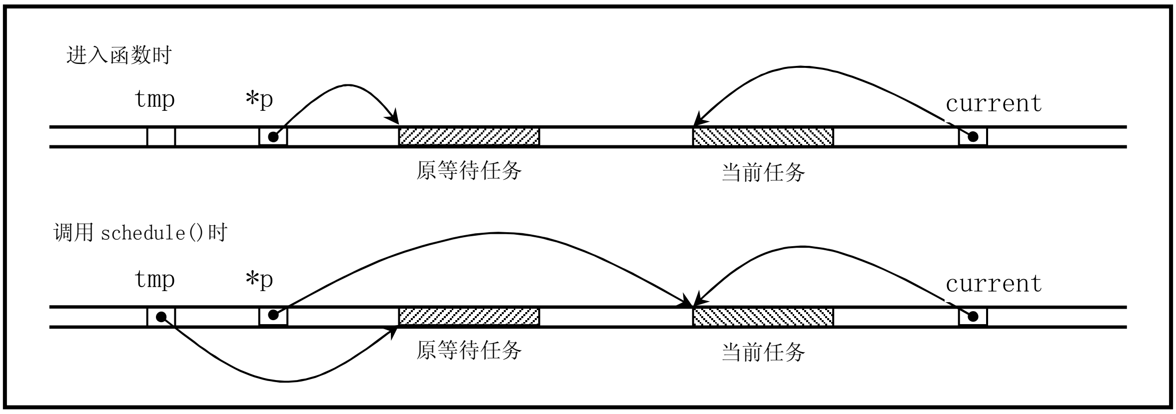


图 5.4 sleep_on()函数中指针变化示意图。

当刚进入该函数时，队列头指针*p 指向已经在等待队列中等待的任务结构（进程描述符）。当然，在系统刚开始执行时，等待队列上无等待任务。因此上图中的原等待任务在刚开始时是不存在的，此时*p 指向 NULL。通过指针操作，在调用调度程序之前，队列头指针指向了当前任务结构，而函数中的临时指针 tmp 指向了原等待任务。从而通过该临时指针的作用，在该函数被嵌套调用时，程序就隐式地构筑出一个等待队列。从下面的图 5.5 中，我们可以更容易地理解 sleep_on()函数的等待队列形成过程。图中示出了当向队列头部插入第三个任务时的情况。

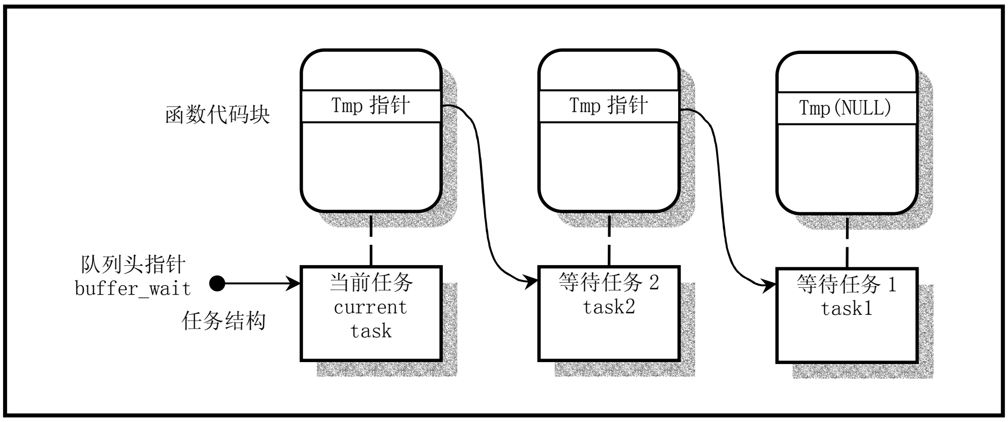


图 5.5 sleep_on()函数的隐式任务等待队列。

另一个函数 interruptible_sleep_on()的结构与 sleep_on()的基本类似，只是在进行调度之前是把当前任务置成了可中断等待状态，并在本任务被唤醒后还需要队列上是否有后来的等待任务，若有则调度它们先运行。在内核 0.12 开始，这两个函数被合二为一，仅用任务的状态作为参数来区分这两种情况。

5.7.3.3 软盘控制器编程

在编程时需要访问 4 个端口，分别对应一个或多个寄存器。对于 1.2M 的软盘控制器有以下一些端口。

表 5.3 软盘控制器端口

I/O 端口	读写性	寄存器名称
0x3f2	只写	数字输出寄存器(数字控制寄存器)
0x3f4	只读	FDC 主状态寄存器
0x3f5	读/写	FDC 数据寄存器

0x3f7	只读	数字输入寄存器
0x3f7	只写	磁盘控制寄存器(传输率控制)

数字输出端口（数字控制端口）是一个 8 位寄存器，它控制驱动器马达开启、驱动器选择、启动/复位 FDC 以及允许/禁止 DMA 及中断请求。

FDC 的主状态寄存器也是一个 8 位寄存器，用于反映软盘控制器 FDC 和软盘驱动器 FDD 的基本状态。通常，在 CPU 向 FDC 发送命令之前或从 FDC 获取操作结果之前，都要读取主状态寄存器的状态位，以判别当前 FDC 数据寄存器是否就绪，以及确定数据传送的方向。

FDC 的数据端口对应多个寄存器（只写型命令寄存器和参数寄存器、只读型结果寄存器），但任一时刻只能有一个寄存器出现在数据端口 0x3f5。在访问只写型寄存器时，主状态控制的 DIO 方向位必须为 0（CPU → FDC），访问只读型寄存器时则反之。在读取结果时只有在 FDC 不忙之后才算读完结果，通常结果数据最多有 7 个字节。

软盘控制器共可以接受 15 条命令。每个命令均经历三个阶段：命令阶段、执行阶段和结果阶段。

命令阶段是 CPU 向 FDC 发送命令字节和参数字节。每条命令的第一个字节总是命令字节（命令码）。其后跟着 0--8 字节的参数。

执行阶段是 FDC 执行命令规定的操作。在执行阶段 CPU 是不加干预的，一般是通过 FDC 发出中断请求获知命令执行的结束。如果 CPU 发出的 FDC 命令是传送数据，则 FDC 可以以中断方式或 DMA 方式进行。中断方式每次传送 1 字节。DMA 方式是在 DMA 控制器管理下，FDC 与内存进行数据的传输直至全部数据传送完。此时 DMA 控制器会将传输字节计数终止信号通知 FDC，最后由 FDC 发出中断请求信号告知 CPU 执行阶段结束。

结果阶段是由 CPU 读取 FDC 数据寄存器返回值，从而获得 FDC 命令执行的结果。返回结果数据的长度为 0--7 字节。对于没有返回结果数据的命令，则应向 FDC 发送检测中断状态命令获得操作的状态。

5.8 signal.c 文件

5.8.1 功能描述

本程序给出了设置和获取进程信号阻塞码（屏蔽码）系统调用函数 sys_ssetmask()和 sys_sgetmask()、信号处理系统调用 sys_singal()、修改进程在收到特定信号时所采取的行动的系统调用 sys_sigaction()以及在系统调用中断处理程序中处理信号的函数 do_signal()。有关信号操作的发送信号函数 send_sig()和通知父进程函数 tell_father()被包含在另一个程序（exit.c）中。程序中的名称前缀 sig 均是信号 signal 的简称。

signal()和 sigaction()的功能比较类似，都是改变信号原处理句柄（handler，或称为处理程序）。但 signal()会返回原信号处理句柄，并且在新句柄被调用一次后句柄就会恢复到默认值。而 sigaction()则可以进行更自由的设置。

do_signal()函数是内核系统调用(int 0x80)中断处理程序中信号的预处理程序。该函数的主要作用是将信号的处理句柄插入到用户程序堆栈中。这样，在当前系统调用结束返回后就会立刻执行信号句柄程序，然后再继续执行用户的程序，见图 5.6 所示。

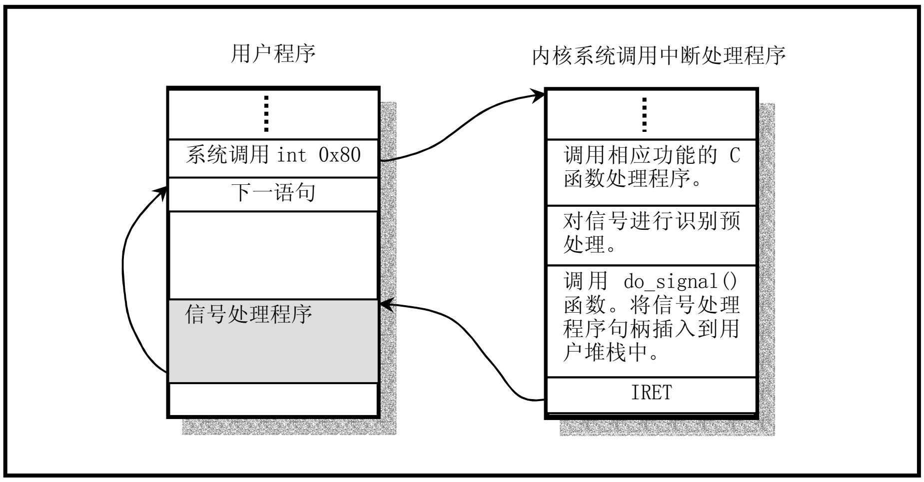


图 5.6 信号处理程序的调用方式。

5.8.2 代码注释

列表 5.8 linux/kernel/signal.c 程序

```
/*
 *  linux/kernel/signal.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

#include <signal.h>       // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。

volatile void do_exit(int error_code);   // 前面的限定符 volatile 要求编译器不要对其进行优化。

// 获取当前任务信号屏蔽位图（屏蔽码）。
int sys_sgetmask()
{
        return current->blocked;
}

// 设置新的信号屏蔽位图。SIGKILL 不能被屏蔽。返回值是原信号屏蔽位图。
int sys_ssetmask(int newmask)
{
        int old=current->blocked;

        current->blocked = newmask & ~(1<<(SIGKILL-1));
        return old;
}

// 复制 sigaction 数据到 fs 数据段 to 处。。
static inline void save_old(char * from,char * to)
```

```
{
        int i;

        verify_area(to, sizeof(struct sigaction));  // 验证 to 处的内存是否足够。
        for (i=0 ; i< sizeof(struct sigaction) ; i++) {
                put_fs_byte(*from,to);              // 复制到 fs 段。一般是用户数据段。
                from++;                       // put_fs_byte()在 include/asm/segment.h 中。
                to++;
        }
}

// 把 sigaction 数据从 fs 数据段 from 位置复制到 to 处。
static inline void get_new(char * from,char * to)
{
        int i;

        for (i=0 ; i< sizeof(struct sigaction) ; i++)
                *(to++) = get_fs_byte(from++);
}

// signal()系统调用。类似于 sigaction()。为指定的信号安装新的信号句柄(信号处理程序)。
// 信号句柄可以是用户指定的函数，也可以是 SIG_DFL（默认句柄）或 SIG_IGN（忽略）。
// 参数 signum --指定的信号；handler -- 指定的句柄；restorer –原程序当前执行的地址位置。
// 函数返回原信号句柄。
int sys_signal(int signum, long handler, long restorer)
{
        struct sigaction tmp;

        if (signum<1 || signum>32 || signum==SIGKILL) // 信号值要在（1-32）范围内，
                return -1;                            // 并且不得是 SIGKILL。
        tmp.sa_handler = (void (*)(int)) handler;     // 指定的信号处理句柄。
        tmp.sa_mask = 0;                              // 执行时的信号屏蔽码。
        tmp.sa_flags = SA_ONESHOT | SA_NOMASK;        // 该句柄只使用 1 次后就恢复到默认值，
                                                      // 并允许信号在自己的处理句柄中收到。
        tmp.sa_restorer = (void (*)(void)) restorer;  // 保存返回地址。
        handler = (long) current->sigaction[signum-1].sa_handler;
        current->sigaction[signum-1] = tmp;
        return handler;
}

// sigaction()系统调用。改变进程在收到一个信号时的操作。signum 是除了 SIGKILL 以外的任何
// 信号。[如果新操作(action)不为空]则新操作被安装。如果 oldaction 指针不为空，则原操作
// 被保留到 oldaction。成功则返回 0，否则为-1。
int sys_sigaction(int signum, const struct sigaction * action,
        struct sigaction * oldaction)
{
        struct sigaction tmp;

// 信号值要在（1-32）范围内，并且信号 SIGKILL 的处理句柄不能被改变。
        if (signum<1 || signum>32 || signum==SIGKILL)
                return -1;
// 在信号的 sigaction 结构中设置新的操作（动作）。
        tmp = current->sigaction[signum-1];
```

```
        get_new((char *) action,
                (char *) (signum-1+current->sigaction));
// 如果 oldaction 指针不为空的话，则将原操作指针保存到 oldaction 所指的位置。
        if (oldaction)
                save_old((char *) &tmp,(char *) oldaction);
// 如果允许信号在自己的信号句柄中收到，则令屏蔽码为 0，否则设置屏蔽本信号。
        if (current->sigaction[signum-1].sa_flags & SA_NOMASK)
                current->sigaction[signum-1].sa_mask = 0;
        else
                current->sigaction[signum-1].sa_mask |= (1<<(signum-1));
        return 0;
}

// 系统调用中断处理程序中真正的信号处理程序（在 kernel/system_call.s,119 行）。
// 该段代码的主要作用是将信号的处理句柄插入到用户程序堆栈中，并在本系统调用结束
// 返回后立刻执行信号句柄程序，然后继续执行用户的程序。
void do_signal(long signr,long eax, long ebx, long ecx, long edx,
        long fs, long es, long ds,
        long eip, long cs, long eflags,
        unsigned long * esp, long ss)
{
        unsigned long sa_handler;
        long old_eip=eip;
        struct sigaction * sa = current->sigaction + signr - 1; //current->sigaction[signu-1]。
        int longs;
        unsigned long * tmp_esp;

        sa_handler = (unsigned long) sa->sa_handler;
// 如果信号句柄为 SIG_IGN(忽略)，则返回；如果句柄为 SIG_DFL(默认处理)，则如果信号是
// SIGCHLD 则返回，否则终止进程的执行
        if (sa_handler==1)
                return;
        if (!sa_handler) {
                if (signr==SIGCHLD)
                        return;
                else
                        do_exit(1<<(signr-1)); // [?? 为什么以信号位图为参数？不为什么!?
☺]
                                               // 这里应该是 do_exit(1<<signr))。
        }
// 如果该信号句柄只需使用一次，则将该句柄置空(该信号句柄已经保存在 sa_handler 指针中)。
        if (sa->sa_flags & SA_ONESHOT)
                sa->sa_handler = NULL;
// 下面这段代码将信号处理句柄插入到用户堆栈中，同时也将 sa_restorer,signr,进程屏蔽码(如果
// SA_NOMASK 没置位),eax,ecx,edx 作为参数以及原调用系统调用的程序返回指针及标志寄存器值
// 压入堆栈。因此在本次系统调用中断(0x80)返回用户程序时会首先执行用户的信号句柄程序，然后
// 再继续执行用户程序。

// 将用户调用系统调用的代码指针 eip 指向该信号处理句柄。
        *(&eip) = sa_handler;
// 如果允许信号自己的处理句柄收到信号自己，则也需要将进程的阻塞码压入堆栈。
        longs = (sa->sa_flags & SA_NOMASK)?7:8;
// 将原调用程序的用户的堆栈指针向下扩展 7（或 8）个长字（用来存放调用信号句柄的参数等），
```

```
// 并检查内存使用情况（例如如果内存超界则分配新页等）。
        *(&esp) -= longs;
        verify_area(esp,longs*4);
// 在用户堆栈中从下到上存放 sa_restorer, 信号 signr, 屏蔽码 blocked(如果 SA_NOMASK 置位),
// eax, ecx, edx, eflags 和用户程序原代码指针。
        tmp_esp=esp;
        put_fs_long((long) sa->sa_restorer,tmp_esp++);
        put_fs_long(signr,tmp_esp++);
        if (!(sa->sa_flags & SA_NOMASK))
                put_fs_long(current->blocked,tmp_esp++);
        put_fs_long(eax,tmp_esp++);
        put_fs_long(ecx,tmp_esp++);
        put_fs_long(edx,tmp_esp++);
        put_fs_long(eflags,tmp_esp++);
        put_fs_long(old_eip,tmp_esp++);
        current->blocked |= sa->sa_mask;  // 进程阻塞码(屏蔽码)添上 sa_mask 中的码位。
}
```

5.8.3 其它信息

5.8.3.1 进程信号说明

进程中的信号是用于进程之间通信的一种简单消息，通常是下表中的一个标号数值，并且不携带任何其它的信息。例如当一个子进程终止或结束时，就会产生一个标号为 17 的 SIGCHILD 信号发送给父进程，以通知父进程有关子进程的当前状态。

关于一个进程如何处理收到的信号，一般有两种做法：一是程序的进程不去处理，此时该信号会由系统相应的默认信号处理程序进行处理；第二种做法是进程使用自己的信号处理程序来处理信号。

表 5.4 进程信号

标号	名称	说明	默认操作
1	SIGHUP	(Hangup) 当你不再控制终端时内核会产生该信号，或者当你关闭 Xterm 或断开 modem。由于后台程序没有控制的终端，因而它们常用 SIGUP 来发出需要重新读取其配置文件的信号。	(Abort) 挂断控制中断或进程。
2	SIGINT	(Interrupt) 来自键盘的终端。通常终端驱动程序会将其与^C 绑定。	(Abort) 终止程序。
3	SIGQUIT	(Quit) 来自键盘的终端。通常终端驱动程序会将其与^\绑定。	(Dump) 程序被终止并产生 dump core 文件。
4	SIGILL	(Illegal Instruction) 程序出错或者执行了一个非法的操作指令。	(Dump) 程序被终止并产生 dump core 文件。
5	SIGTRAP	(Breakpoint/Trace Trap) 调试用，跟踪断点。	
6	SIGABRT	(Abort) 放弃执行，异常结束。	(Dump) 程序被终止并产生 dump core 文件。
6	SIGIOT	(IO Trap) 同 SIGABRT	(Dump) 程序被终止并产生 dump core 文件。
7	SIGUNUSED	(Unused) 没有使用。	
8	SIGFPE	(Floating Point Exception) 浮点异常。	(Dump) 程序被终止并产生 dump core 文件。

9	SIGKILL	(Kill) 程序被终止。该信号不能被捕获或者被忽略。想立刻终止一个进程，就发送信号 9。注意程序将没有任何机会做清理工作。	(Abort) 程序被终止。
10	SIGUSR1	(User defined Signal 1) 用户定义的信号。	(Abort) 进程被终止。
11	SIGSEGV	(Segmentation Violation) 当程序引用无效的内存时会产生此信号。比如：寻址没有映射的内存；寻址未许可的内存。	(Dump) 程序被终止并产生 dump core 文件。
12	SIGUSR2	(User defined Signal 2) 保留给用户程序用于 IPC 或其它目的。	(Abort) 进程被终止。
13	SIGPIPE	(Pipe) 当程序向一个套接字或管道写时由于没有读者而产生该信号。	(Abort) 进程被终止。
14	SIGALRM	(Alarm) 该信号会在用户调用 alarm 系统调用所设置的延迟秒数到后产生。该信号常用判别于系统调用超时。	(Abort) 进程被终止。
15	SIGTERM	(Terminate) 用于和善地要求一个程序终止。它是 kill 的默认信号。与 SIGKILL 不同，该信号能被捕获，这样就能在退出运行前做清理工作。	(Abort) 进程被终止。
16	SIGSTKFLT	(Stack fault on coprocessor) 协处理器堆栈错误。	(Abort) 进程被终止。
17	SIGCHLD	(Child) 父进程发出。停止或终止子进程。可改变其含义挪作它用。	(Ignore) 子进程停止或结束。
18	SIGCONT	(Continue) 该信号致使被 SIGSTOP 停止的进程恢复运行。可以被捕获。	(Continue) 恢复进程的执行。
19	SIGSTOP	(Stop) 停止进程的运行。该信号不可被捕获或忽略。	(Stop) 停止进程运行。
20	SIGTSTP	(Terminal Stop) 向终端发送停止键序列。该信号可以被捕获或忽略。	(Stop) 停止进程运行。
21	SIGTTIN	(Terminal Input on Background) 后台进程试图从一个不再被控制的终端上读取数据，此时该进程将被停止，直到收到 SIGCONT 信号。该信号可以被捕获或忽略。	(Stop) 停止进程运行。
22	SIGTTOU	(TTY Output on Background) 后台进程试图向一个不再被控制的终端上输出数据，此时该进程将被停止，直到收到 SIGCONT 信号。该信号可被捕获或忽略。	(Stop) 停止进程运行。

5.9 exit.c 文件

5.9.1 功能描述

该程序主要描述了进程（任务）终止和退出的处理事宜。主要包含进程释放、会话（进程组）终止和程序退出处理函数以及杀死进程、终止进程、挂起进程等系统调用函数。还包括进程信号发送函数 send_sig()和通知父进程子进程终止的函数 tell_father()。

释放进程的函数 release()主要根据指定的任务数据结构（任务描述符）指针，在任务数组中删除指定的进程指针、释放相关内存页并立刻让内核重新调度任务的运行。

进程组终止函数 kill_session()通过向会话号与当前进程相同的进程发送挂断进程的信号。

系统调用 sys_kill()用于向进程发送任何指定的信号。根据参数 pid（进程标识号）的数值的不同，该系统调用会向不同的进程或进程组发送信号。程序注释中已经列出了各种不同情况的处理方式。

程序退出处理函数 do_exit()是在系统调用的中断处理程序中被调用的。它首先会释放当前进程的代码段和数据段所占的内存页面，然后向子进程发送终止信号 SIGCHLD。接着关闭当前进程打开的所有文件、释放使用的终端设备、协处理器设备，若当前进程是进程组的领头进程，则还需要终止所有相关进程。随后把当前进程置为僵死状态，设置退出码，并向其父进程发送子进程终止信号。最后让内核重新调度任务的运行。

系统调用 waitpid()用于挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止该进程的信号，或者是需要调用一个信号句柄（信号处理程序）。如果 pid 所指的子进程早已退出（已成所谓的僵死进程），则本调用将立刻返回。子进程使用的所有资源将释放。该函数的具体操作也要根据其参数进行不同的处理。详见代码中的相关注释。

5.9.2 代码注释

列表 5.9 linux/kernel/exit.c 程序

```
/*
 *  linux/kernel/exit.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)
#include <signal.h>         // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。
#include <sys/wait.h>       // 等待调用头文件。定义系统调用 wait()和 waitpid()及相关常数符号。

#include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                            // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
#include <linux/tty.h>      // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
#include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

int sys_pause(void);
int sys_close(int fd);

//// 释放指定进程(任务)。
void release(struct task_struct * p)
{
        int i;

        if (!p)
                return;
        for (i=1 ; i<NR_TASKS ; i++)          // 扫描任务数组，寻找指定任务。
                if (task[i]==p) {
                        task[i]=NULL;         // 置空该任务项并释放相关内存页。
                        free_page((long)p);
                        schedule();           // 重新调度。
                        return;
                }
        panic("trying to release non-existent task");  // 指定任务若不存在则死机。
}
```

```
//// 向指定任务(*p)发送信号(sig)，权限为 priv。
static inline int send_sig(long sig,struct task_struct * p,int priv)
{
// 若信号不正确或任务指针为空则出错退出。
        if (!p || sig<1 || sig>32)
                return -EINVAL;
// 若有权或进程有效用户标识符(euid)就是指定进程的 euid 或者是超级用户，则在进程位图中添加
// 该信号，否则出错退出。其中 suser()定义为(current->euid==0)，用于判断是否超级用户。
        if (priv || (current->euid==p->euid) || suser())
                p->signal |= (1<<(sig-1));
        else
                return -EPERM;
        return 0;
}

//// 终止会话(session)。
static void kill_session(void)
{
        struct task_struct **p = NR_TASKS + task;  // 指针*p 首先指向任务数组最末端。

// 对于所有的任务（除任务 0 以外），如果其会话等于当前进程的会话就向它发送挂断进程信号。
        while (--p > &FIRST_TASK) {
                if (*p && (*p)->session == current->session)
                        (*p)->signal |= 1<<(SIGHUP-1);   // 发送挂断进程信号。
        }
}

/*
 * XXX need to check permissions needed to send signals to process
 * groups, etc. etc.  kill() permissions semantics are tricky!
 */
/*
 * 为了向进程组等发送信号，XXX 需要检查许可。kill()的许可机制非常巧妙!
 * /
//// kill()系统调用可用于向任何进程或进程组发送任何信号。
// 如果 pid 值>0，则信号被发送给 pid。
// 如果 pid=0，那么信号就会被发送给当前进程的进程组中的所有进程。
// 如果 pid=-1，则信号 sig 就会发送给除第一个进程外的所有进程。
// 如果 pid < -1，则信号 sig 将发送给进程组-pid 的所有进程。
// 如果信号 sig 为 0，则不发送信号，但仍会进行错误检查。如果成功则返回 0。
int sys_kill(int pid,int sig)
{
        struct task_struct **p = NR_TASKS + task;
        int err, retval = 0;

        if (!pid) while (--p > &FIRST_TASK) {
                if (*p && (*p)->pgrp == current->pid)
                        if (err=send_sig(sig,*p,1))
                                retval = err;
        } else if (pid>0) while (--p > &FIRST_TASK) {
                if (*p && (*p)->pid == pid)
                        if (err=send_sig(sig,*p,0))
```

```
                                retval = err;
        } else if (pid == -1) while (--p > &FIRST_TASK)
                if (err = send_sig(sig,*p,0))
                        retval = err;
        else while (--p > &FIRST_TASK)
                if (*p && (*p)->pgrp == -pid)
                        if (err = send_sig(sig,*p,0))
                                retval = err;
        return retval;
}

//// 通知父进程 -- 向进程 pid 发送信号 SIGCHLD：子进程将停止或终止。
// 如果没有找到父进程，则自己释放。
static void tell_father(int pid)
{
        int i;

        if (pid)
                for (i=0;i<NR_TASKS;i++) {
                        if (!task[i])
                                continue;
                        if (task[i]->pid != pid)
                                continue;
                        task[i]->signal |= (1<<(SIGCHLD-1));
                        return;
                }
/* if we don't find any fathers, we just release ourselves */
/* This is not really OK. Must change it to make father 1 */
        printk("BAD BAD - no father found\n\r");
        release(current);  // 如果没有找到父进程，则自己释放。
}

//// 程序退出处理程序。在系统调用的中断处理程序中被调用。
int do_exit(long code)    // code 是错误码。
{
        int i;

// 释放当前进程代码段和数据段所占的内存页(free_page_tables()在 mm/memory.c,105 行)。
        free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
        free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
// 如果当前进程有子进程，就将子进程的 father 置为 1(其父进程改为进程 1)。如果该子进程已经
// 处于僵死(ZOMBIE)状态，则向进程 1 发送子进程终止信号 SIGCHLD。
        for (i=0 ; i<NR_TASKS ; i++)
                if (task[i] && task[i]->father == current->pid) {
                        task[i]->father = 1;
                        if (task[i]->state == TASK_ZOMBIE)
                                /* assumption task[1] is always init */
                                (void) send_sig(SIGCHLD, task[1], 1);
                }
// 关闭当前进程打开着的所有文件。
        for (i=0 ; i<NR_OPEN ; i++)
                if (current->filp[i])
                        sys_close(i);
```

```
// 对当前进程工作目录 pwd、根目录 root 以及运行程序的 i 节点进行同步操作，并分别置空。
        iput(current->pwd);
        current->pwd=NULL;
        iput(current->root);
        current->root=NULL;
        iput(current->executable);
        current->executable=NULL;
// 如果当前进程是领头(leader)进程并且其有控制的终端，则释放该终端。
        if (current->leader && current->tty >= 0)
                tty_table[current->tty].pgrp = 0;
// 如果当前进程上次使用过协处理器，则将 last_task_used_math 置空。
        if (last_task_used_math == current)
                last_task_used_math = NULL;
// 如果当前进程是 leader 进程，则终止所有相关进程。
        if (current->leader)
                kill_session();
// 把当前进程置为僵死状态，并设置退出码。
        current->state = TASK_ZOMBIE;
        current->exit_code = code;
// 通知父进程，也即向父进程发送信号 SIGCHLD -- 子进程将停止或终止。
        tell_father(current->father);
        schedule();       // 重新调度进程的运行。
        return (-1);     /* just to suppress warnings */
}

//// 系统调用 exit()。终止进程。
int sys_exit(int error_code)
{
        return do_exit((error_code&0xff)<<8);
}

//// 系统调用 waitpid()。挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止
// 该进程的信号，或者是需要调用一个信号句柄（信号处理程序）。如果 pid 所指的子进程早已
// 退出（已成所谓的僵死进程），则本调用将立刻返回。子进程使用的所有资源将释放。
// 如果 pid > 0, 表示等待进程号等于 pid 的子进程。
// 如果 pid = 0, 表示等待进程组号等于当前进程的任何子进程。
// 如果 pid < -1, 表示等待进程组号等于 pid 绝对值的任何子进程。
// [ 如果 pid = -1, 表示等待任何子进程。]
// 若 options = WUNTRACED，表示如果子进程是停止的，也马上返回。
// 若 options = WNOHANG，表示如果没有子进程退出或终止就马上返回。
// 如果 stat_addr 不为空，则就将状态信息保存到那里。
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
        int flag, code;
        struct task_struct ** p;

        verify_area(stat_addr,4);
repeat:
        flag=0;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p) {  // 从任务数组末端开始扫描所有任务。
                if (!*p || *p == current)              // 跳过空项和本进程项。
                        continue;
                if ((*p)->father != current->pid)      // 如果不是当前进程的子进程则跳过。
```

```
                        continue;
                if (pid>0) {                        // 如果指定的 pid>0，但扫描的进程 pid
                        if ((*p)->pid != pid)       // 与之不等，则跳过。
                                continue;
                } else if (!pid) {                  // 如果指定的 pid=0，但扫描的进程组号
                        if ((*p)->pgrp != current->pgrp) // 与当前进程的组号不等，则跳过。
                                continue;
                } else if (pid != -1) {             // 如果指定的 pid<-1，但扫描的进程组号
                        if ((*p)->pgrp != -pid)     // 与其绝对值不等，则跳过。
                                continue;
                }
                switch ((*p)->state) {
                        case TASK_STOPPED:
                                if (!(options & WUNTRACED))
                                        continue;
                                put_fs_long(0x7f,stat_addr);   // 置状态信息为 0x7f。
                                return (*p)->pid;              // 退出,返回子进程的进程号。
                        case TASK_ZOMBIE:
                                current->cutime += (*p)->utime; // 更新当前进程的子进程用户
                                current->cstime += (*p)->stime; // 态和核心态运行时间。
                                flag = (*p)->pid;
                                code = (*p)->exit_code;        // 取子进程的退出码。
                                release(*p);                   // 释放该子进程。
                                put_fs_long(code,stat_addr);   // 置状态信息为退出码值。
                                return flag;                   // 退出，返回子进程的 pid.
                        default:
                                flag=1;         // 如果子进程不在停止或僵死状态，则 flag=1。
                                continue;
                }
        }
        if (flag) {                                 // 如果子进程没有处于退出或僵死状态，
                if (options & WNOHANG)              // 并且 options = WNOHANG，则立刻返回。
                        return 0;
                current->state=TASK_INTERRUPTIBLE;  // 置当前进程为可中断等待状态。
                schedule();                         // 重新调度。
                if (!(current->signal &= ~(1<<(SIGCHLD-1)))) // 又开始执行本进程时，
                        goto repeat;    // 如果进程没有收到除 SIGCHLD 的信号，则还是重复处理。
                else
                        return -EINTR;      // 退出，返回出错码。
        }
        return -ECHILD;
}
```

5.10 fork.c 程序

5.10.1 功能描述

fork()系统调用用于创建子进程。Linux 中所有进程都是进程 0（任务 0）的子进程。该程序是 sys_fork()（在 kernel/system_call.s 中定义）系统调用的辅助处理函数集，给出了 sys_fork()系统调用中使用的两个 C 语言函数：find_empty_process()和 copy_process()。还包括进程内存区域验证与内存分配函数 verify_area()。

copy_process()是用于创建并复制进程的代码段和数据段以及环境。在进程复制过程中，主要牵涉到进程数据结构中信息的设置。

5.10.2 代码注释

列表 5.10 linux/kernel/fork.c 程序

```
/*
 *  linux/kernel/fork.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *  'fork.c' contains the help-routines for the 'fork' system call
 * (see also system_call.s), and some misc functions ('verify_area').
 * Fork is rather simple, once you get the hang of it, but the memory
 * management can be a bitch. See 'mm/mm.c': 'copy_page_tables()'
 */
/*
 * 'fork.c'中含有系统调用'fork'的辅助子程序（参见 system_call.s），以及一些其它函数
 * ('verify_area')。一旦你了解了 fork，就会发现它是非常简单的，但内存管理却有些难度。
 * 参见'mm/mm.c'中的'copy_page_tables()'。
 */
#include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。

#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
#include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

extern void write_verify(unsigned long address);

long last_pid=0;

// 验证内存区域。对于给定起始地址和大小的内存块进行检查，如果发现超出分配给进程的页，就
// 再另行分配新内存页。
void verify_area(void * addr,int size)
{
        unsigned long start;

```

```
        start = (unsigned long) addr;
        size += start & 0xfff;
        start &= 0xfffff000;
        start += get_base(current->ldt[2]);
        while (size>0) {
                size -= 4096;
                write_verify(start);
                start += 4096;
        }
}

// 设置新任务的代码和数据段基址、限长并复制页表。
// nr 为新任务号；p 是新任务数据结构的指针。
int copy_mem(int nr,struct task_struct * p)
{
        unsigned long old_data_base,new_data_base,data_limit;
        unsigned long old_code_base,new_code_base,code_limit;

        code_limit=get_limit(0x0f);  // 取局部描述符表中代码段描述符项中段限长。
        data_limit=get_limit(0x17);  // 取局部描述符表中数据段描述符项中段限长。
        old_code_base = get_base(current->ldt[1]);  // 取原代码段基址。
        old_data_base = get_base(current->ldt[2]);  // 取原数据段基址。
        if (old_data_base != old_code_base)    // 0.11 版不支持代码和数据段分立的情况。
                panic("We don't support separate I&D");
        if (data_limit < code_limit)           // 如果数据段长度 < 代码段长度也不对。
                panic("Bad data_limit");
        new_data_base = new_code_base = nr * 0x4000000; // 新基址=任务号*64Mb(任务大小)。
        p->start_code = new_code_base;
        set_base(p->ldt[1],new_code_base);  // 设置代码段描述符中基址域。
        set_base(p->ldt[2],new_data_base);  // 设置数据段描述符中基址域。
        if (copy_page_tables(old_data_base,new_data_base,data_limit)) {// 复制代码和数据段。
                free_page_tables(new_data_base,data_limit); // 如果出错则释放申请的内存。
                return -ENOMEM;
        }
        return 0;
}

/*
 *  Ok, this is the main fork-routine. It copies the system process
 * information (task[nr]) and sets up the necessary registers. It
 * also copies the data segment in it's entirety.
 */
/*
 * OK，下面是主要的 fork 子程序。它复制系统进程信息(task[n])并且设置必要的寄存器。
 * 它还整个地复制数据段。
 */
// 复制进程。
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
                long ebx,long ecx,long edx,
                long fs,long es,long ds,
                long eip,long cs,long eflags,long esp,long ss)
{
        struct task_struct *p;
```

```
        int i;
        struct file *f;

        p = (struct task_struct *) get_free_page();    // 为新任务数据结构分配内存。
        if (!p)                          // 如果内存分配出错，则返回出错码并退出。
                return -EAGAIN;
        task[nr] = p;                    // 将新任务结构指针放入任务数组中。
                                         // 其中 nr 为任务号，由前面 find_empty_process()返回。
        *p = *current;  /* NOTE! this doesn't copy the supervisor stack */
                        /* 注意！这样做不会复制超级用户的堆栈 */（只复制当前进程内容）。
        p->state = TASK_UNINTERRUPTIBLE; // 将新进程的状态先置为不可中断等待状态。
        p->pid = last_pid;               // 新进程号。由前面调用 find_empty_process()得到。
        p->father = current->pid;        // 设置父进程号。
        p->counter = p->priority;
        p->signal = 0;                   // 信号位图置 0。
        p->alarm = 0;
        p->leader = 0;          /* process leadership doesn't inherit */
                                /* 进程的领导权是不能继承的 */
        p->utime = p->stime = 0;         // 初始化用户态时间和核心态时间。
        p->cutime = p->cstime = 0;       // 初始化子进程用户态和核心态时间。
        p->start_time = jiffies;         // 当前滴答数时间。
// 以下设置任务状态段 TSS 所需的数据（参见列表后说明）。
        p->tss.back_link = 0;
        p->tss.esp0 = PAGE_SIZE + (long) p;   // 堆栈指针（由于是给任务结构 p 分配了 1 页
                                              // 新内存，所以此时 esp0 正好指向该页顶端）。
        p->tss.ss0 = 0x10;               // 堆栈段选择符（内核数据段）[??]。
        p->tss.eip = eip;                // 指令代码指针。
        p->tss.eflags = eflags;          // 标志寄存器。
        p->tss.eax = 0;
        p->tss.ecx = ecx;
        p->tss.edx = edx;
        p->tss.ebx = ebx;
        p->tss.esp = esp;
        p->tss.ebp = ebp;
        p->tss.esi = esi;
        p->tss.edi = edi;
        p->tss.es = es & 0xffff;         // 段寄存器仅 16 位有效。
        p->tss.cs = cs & 0xffff;
        p->tss.ss = ss & 0xffff;
        p->tss.ds = ds & 0xffff;
        p->tss.fs = fs & 0xffff;
        p->tss.gs = gs & 0xffff;
        p->tss.ldt = _LDT(nr);   // 该新任务 nr 的局部描述符表选择符（LDT 的描述符在 GDT 中）。
        p->tss.trace_bitmap = 0x80000000;  （高 16 位有效）。
// 如果当前任务使用了协处理器，就保存其上下文。
        if (last_task_used_math == current)
                __asm__("clts ; fnsave %0"::"m" (p->tss.i387));
// 设置新任务的代码和数据段基址、限长并复制页表。如果出错（返回值不是 0），则复位任务数组中
// 相应项并释放为该新任务分配的内存页。
        if (copy_mem(nr,p)) {              // 返回不为 0 表示出错。
                task[nr] = NULL;
                free_page((long) p);
                return -EAGAIN;
```

```
        }
// 如果父进程中有文件是打开的，则将对应文件的打开次数增 1。
        for (i=0; i<NR_OPEN;i++)
                if (f=p->filp[i])
                        f->f_count++;
// 将当前进程（父进程）的 pwd, root 和 executable 引用次数均增 1。
        if (current->pwd)
                current->pwd->i_count++;
        if (current->root)
                current->root->i_count++;
        if (current->executable)
                current->executable->i_count++;
// 设置新任务的 TSS 和 LDT 描述符项（在 GDT 中），数据从 task 结构中取。
// 在任务切换时，任务寄存器 tr 由 CPU 自动加载。局部描述符表寄存器 ldtr 已在 task0 时加载。
        set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
        set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
        p->state = TASK_RUNNING;        /* do this last, just in case */
                                        /* 最后再将新任务设置成可运行状态，以防万一 */
        return last_pid;                // 返回新进程号（与任务号是不同的）。
}

// 为新进程取得不重复的进程号 last_pid，并返回在任务数组中的任务号(数组 index)。
int find_empty_process(void)
{
        int i;

        repeat:
                if ((++last_pid)<0) last_pid=1;
                for(i=0 ; i<NR_TASKS ; i++)
                        if (task[i] && task[i]->pid == last_pid) goto repeat;
        for(i=1 ; i<NR_TASKS ; i++)     // 任务 0 排除在外。
                if (!task[i])
                        return i;
        return -EAGAIN;
}

```

5.10.3 其它信息

5.10.3.1 任务状态段（TSS）信息

下面图 5.7 是任务状态段 TSS（Task State Segment）的内容。对它的说明请参见附录。

31 … 23 … 16	15 … 7 … 0	
I/O 映射图基地址(MAP BASE)	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	64
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	局部描述符表(LDT)的选择符	60
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	GS	5C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	FS	58
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DS	54
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS	50
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	CS	4C

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ES	48
EDI		44
ESI		40
EBP		3C
ESP		38
EBX		34
EDX		30
ECX		2C
EAX		28
EFLAGS		24
指令指针(EIP)		20
页目录基地址寄存器 CR3（PDBR）		1C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS2	18
ESP2		14
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS1	10
ESP1		0C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS0	08
ESP0		04
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	前一执行任务 TSS 的描述符	00

图 5.7 任务状态段 TSS 中的信息。

CPU 管理任务需要的所有信息被存储于一个特殊类型的段中，任务状态段(task state segment - TSS)。图中显示出执行 80386 任务的 TSS 格式。

TSS 中的字段可以分为两类：

1. CPU 在进行任务切换时更新的动态信息集。这些字段有：
   - o 通用寄存器（EAX，ECX，EDX，EBX，ESP，EBP，ESI，EDI)；
   - o 段寄存器（ES, CS, SS, DS, FS, GS)；
   - o 标志寄存器（EIP)；
   - o 指令指针（EIP)；

   前一个执行任务的 TSS 的选择符（仅当返回时才更新)。
2. CPU 读取但不会更改的静态信息集。这些字段有：
   - o 任务的 LDT 的选择符；
   - o 含有任务页目录基地址的寄存器（PDBR)；
   - o 特权级 0-2 的堆栈指针；
   - o 当任务进行切换时导致 CPU 产生一个调试(debug)异常的 T-比特位（调试跟踪位)；
   - o I/O 比特位图基地址（其长度上限就是 TSS 的长度上限，在 TSS 描述符中说明)。

任务状态段可以存放在线形空间的任何地方。与其它各类段相似，任务状态段也是由描述符来定义的。当前正在执行任务的 TSS 是由任务寄存器（TR）来指示的。指令 LTR 和 STR 用来修改和读取任务寄存器中的选择符（任务寄存器的可见部分)。

I/O 比特位图中的每 1 比特对应 1 个 I/O 端口。比如端口 41 的比特位就是 I/O 位图基地址+5，位偏移 1 处。在保护模式中，当遇到 1 个 I/O 指令时(IN, INS, OUT, OUTS)，CPU 首先就会检查当前特权级

是否小于标志寄存器的 IOPL，如果这个条件满足，就执行该 I/O 操作。如果不满足，那么 CPU 就会检查 TSS 中的 I/O 比特位图。如果相应比特位是置位的，就会产生一般保护性异常，否则就会执行该 I/O 操作。

5.11 sys.c 程序

5.11.1 功能描述

sys.c 程序主要包含有很多系统调用功能的实现函数。其中，若返回值为-ENOSYS，则表示本版的 linux 还没有实现该功能，可以参考目前的代码来了解它们的实现方法。所有系统调用的功能说明请参见头文件 include/linux/sys.h。

5.11.2 代码注释

列表 5.11 linux/kernel/sys.c 程序

```
/*
 *  linux/kernel/sys.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。

#include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                            // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/tty.h>      // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
#include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
#include <sys/times.h>      // 定义了进程中运行时间的结构 tms 以及 times()函数原型。
#include <sys/utsname.h>    // 系统名称结构头文件。

// 返回日期和时间。
int sys_ftime()
{
        return -ENOSYS;
}

//
int sys_break()
{
        return -ENOSYS;
}

// 用于当前进程对子进程进行调试(degugging)。
int sys_ptrace()
{
        return -ENOSYS;
}
```

```
// 改变并打印终端行设置。
int sys_stty()
{
        return -ENOSYS;
}

// 取终端行设置信息。
int sys_gtty()
{
        return -ENOSYS;
}

// 修改文件名。
int sys_rename()
{
        return -ENOSYS;
}

//
int sys_prof()
{
        return -ENOSYS;
}

// 设置当前任务的实际以及/或者有效组 ID（gid）。如果任务没有超级用户特权，
// 那么只能互换其实际组 ID 和有效组 ID。如果任务具有超级用户特权，就能任意设置有效的和实际
// 的组 ID。保留的 gid（saved gid）被设置成与有效 gid 同值。
int sys_setregid(int rgid, int egid)
{
        if (rgid>0) {
                if ((current->gid == rgid) ||
                    suser())
                        current->gid = rgid;
                else
                        return(-EPERM);
        }
        if (egid>0) {
                if ((current->gid == egid) ||
                    (current->egid == egid) ||
                    (current->sgid == egid) ||
                    suser())
                        current->egid = egid;
                else
                        return(-EPERM);
        }
        return 0;
}

// 设置进程组号(gid)。如果任务没有超级用户特权，它可以使用 setgid()将其有效 gid
// （effective gid）设置为成其保留 gid(saved gid)或其实际 gid(real gid)。如果任务有
// 超级用户特权，则实际 gid、有效 gid 和保留 gid 都被设置成参数指定的 gid。
int sys_setgid(int gid)
```

```
{
        return(sys_setregid(gid, gid));
}

// 打开或关闭进程计帐功能。
int sys_acct()
{
        return -ENOSYS;
}

// 映射任意物理内存到进程的虚拟地址空间。
int sys_phys()
{
        return -ENOSYS;
}

int sys_lock()
{
        return -ENOSYS;
}

int sys_mpx()
{
        return -ENOSYS;
}

int sys_ulimit()
{
        return -ENOSYS;
}

// 返回从 1970 年 1 月 1 日 00:00:00 GMT 开始计时的时间值（秒）。如果 tloc 不为 null，则时间值
// 也存储在那里。
int sys_time(long * tloc)
{
        int i;

        i = CURRENT_TIME;
        if (tloc) {
                verify_area(tloc,4);    // 验证内存容量是否够（这里是 4 字节）。
                put_fs_long(i,(unsigned long *)tloc);   // 也放入用户数据段 tloc 处。
        }
        return i;
}

/*
 * Unprivileged users may change the real user id to the effective uid
 * or vice versa.
 */
/*
 * 无特权的用户可以见实际用户标识符(real uid)改成有效用户标识符(effective uid)，反之也然。
 */
// 设置任务的实际以及/或者有效用户 ID（uid）。如果任务没有超级用户特权，那么只能互换其
```

```
// 实际用户 ID 和有效用户 ID。如果任务具有超级用户特权，就能任意设置有效的和实际的用户 ID。
// 保留的 uid（saved uid）被设置成与有效 uid 同值。
int sys_setreuid(int ruid, int euid)
{
        int old_ruid = current->uid;

        if (ruid>0) {
                if ((current->euid==ruid) ||
                    (old_ruid == ruid) ||
                    suser())
                        current->uid = ruid;
                else
                        return(-EPERM);
        }
        if (euid>0) {
                if ((old_ruid == euid) ||
                    (current->euid == euid) ||
                    suser())
                        current->euid = euid;
                else {
                        current->uid = old_ruid;
                        return(-EPERM);
                }
        }
        return 0;
}

// 设置任务用户号(uid)。如果任务没有超级用户特权，它可以使用 setuid()将其有效 uid
// （effective uid）设置成其保留 uid(saved uid)或其实际 uid(real uid)。如果任务有
// 超级用户特权，则实际 uid、有效 uid 和保留 uid 都被设置成参数指定的 uid。
int sys_setuid(int uid)
{
        return(sys_setreuid(uid, uid));
}

// 设置系统时间和日期。参数 tptr 是从 1970 年 1 月 1 日 00:00:00 GMT 开始计时的时间值（秒）。
// 调用进程必须具有超级用户权限。
int sys_stime(long * tptr)
{
        if (!suser())              // 如果不是超级用户则出错返回（许可）。
                return -EPERM;
        startup_time = get_fs_long((unsigned long *)tptr) - jiffies/HZ;
        return 0;
}

// 获取当前任务时间。tms 结构中包括用户时间、系统时间、子进程用户时间、子进程系统时间。
int sys_times(struct tms * tbuf)
{
        if (tbuf) {
                verify_area(tbuf,sizeof *tbuf);
                put_fs_long(current->utime,(unsigned long *)&tbuf->tms_utime);
                put_fs_long(current->stime,(unsigned long *)&tbuf->tms_stime);
                put_fs_long(current->cutime,(unsigned long *)&tbuf->tms_cutime);
```

```
163                 put_fs_long(current->cstime,(unsigned long *)&tbuf->tms_cstime);
164         }
165         return jiffies;
166 }
167
    // 当参数 end_data_seg 数值合理，并且系统确实有足够的内存，而且进程没有超越其最大数据段大小
    // 时，该函数设置数据段末尾为 end_data_seg 指定的值。该值必须大于代码结尾并且要小于堆栈
    // 结尾 16KB。返回值是数据段的新结尾值（如果返回值与要求值不同，则表明有错发生）。
    // 该函数并不被用户直接调用，而由 libc 库函数进行包装，并且返回值也不一样。
168 int sys_brk(unsigned long end_data_seg)
169 {
170         if (end_data_seg >= current->end_code &&          // 如果参数>代码结尾，并且
171             end_data_seg < current->start_stack - 16384)  // 小于堆栈-16KB，
172                 current->brk = end_data_seg;              // 则设置新数据段结尾值。
173         return current->brk;                  // 返回进程当前的数据段结尾值。
174 }
175
176 /*
177  * This needs some heave checking ...
178  * I just haven't get the stomach for it. I also don't fully
179  * understand sessions/pgrp etc. Let somebody who does explain it.
180  */
    /*
     * 下面代码需要某些严格的检查…
     * 我只是没有胃口来做这些。我也不完全明白 sessions/pgrp 等。还是让了解它们的人来做吧。
     */
    // 将参数 pid 指定进程的进程组 ID 设置成 pgid。如果参数 pid=0，则使用当前进程号。如果
    // pgid 为 0，则使用参数 pid 指定的进程的组 ID 作为 pgid。如果该函数用于将进程从一个
    // 进程组移到另一个进程组，则这两个进程组必须属于同一个会话(session)。在这种情况下，
    // 参数 pgid 指定了要加入的现有进程组 ID，此时该组的会话 ID 必须与将要加入进程的相同(193 行)。
181 int sys_setpgid(int pid, int pgid)
182 {
183         int i;
184
185         if (!pid)                           // 如果参数 pid=0，则使用当前进程号。
186                 pid = current->pid;
187         if (!pgid)                          // 如果 pgid 为 0，则使用当前进程 pid 作为 pgid。
188                 pgid = current->pid;        // [??这里与 POSIX 的描述有出入]
189         for (i=0 ; i<NR_TASKS ; i++)        // 扫描任务数组，查找指定进程号的任务。
190                 if (task[i] && task[i]->pid==pid) {
191                         if (task[i]->leader)          // 如果该任务已经是首领，则出错返回。
192                                 return -EPERM;
193                         if (task[i]->session != current->session) // 如果该任务的会话 ID
194                                 return -EPERM;        // 与当前进程的不同，则出错返回。
195                         task[i]->pgrp = pgid;         // 设置该任务的 pgrp。
196                         return 0;
197                 }
198         return -ESRCH;
199 }
200
    // 返回当前进程的组号。与 getpgid(0)等同。
201 int sys_getpgrp(void)
202 {
```

```
        return current->pgrp;
}

// 创建一个会话(session)（即设置其 leader=1），并且设置其会话=其组号=其进程号。
int sys_setsid(void)
{
        if (current->leader && !suser())   // 如果当前进程已是会话首领并且不是超级用户
                return -EPERM;              // 则出错返回。
        current->leader = 1;                // 设置当前进程为新会话首领。
        current->session = current->pgrp = current->pid; // 设置本进程 session = pid。
        current->tty = -1;                  // 表示当前进程没有控制终端。
        return current->pgrp;               // 返回会话 ID。
}

// 获取系统信息。其中 utsname 结构包含 5 个字段，分别是：本版本操作系统的名称、网络节点名称、
// 当前发行级别、版本级别和硬件类型名称。
int sys_uname(struct utsname * name)
{
        static struct utsname thisname = {  // 这里给出了结构中的信息，这种编码肯定会改变。
                "linux .0","nodename","release ","version ","machine "
        };
        int i;

        if (!name) return -ERROR;          // 如果存放信息的缓冲区指针为空则出错返回。
        verify_area(name,sizeof *name);    // 验证缓冲区大小是否超限（超出已分配的内存等）。
        for(i=0;i<sizeof *name;i++)        // 将 utsname 中的信息逐字节复制到用户缓冲区中。
                put_fs_byte(((char *) &thisname)[i],i+(char *) name);
        return 0;
}

// 设置当前进程创建文件属性屏蔽码为 mask & 0777。并返回原屏蔽码。
int sys_umask(int mask)
{
        int old = current->umask;

        current->umask = mask & 0777;
        return (old);
}
```

5.12 vsprintf.c 程序

5.12.1 功能描述

主要包括 vsprintf()函数，用于对参数产生格式化的输出。由于该函数是 C 函数库中的标准函数，基本没有涉及内核工作原理，因此可以跳过。直接阅读代码后对该函数的使用说明。

5.12.2 代码注释

列表 5.12 linux/kernel/vsprintf.c 程序

```
/*
 *  linux/kernel/vsprintf.c
 *
 *  (C) 1991  Linus Torvalds
 */

/* vsprintf.c -- Lars Wirzenius & Linus Torvalds. */
/*
 * Wirzenius wrote this portably, Torvalds fucked it up :-)
 */

#include <stdarg.h>       // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了-个
                          // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于
                          // vsprintf、vprintf、vfprintf 函数。
#include <string.h>       // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。

/* we use this so that we can do without the ctype library */
/* 我们使用下面的定义，这样我们就可以不使用 ctype 库了 */
#define is_digit(c)     ((c) >= '' && (c) <= '9')    // 判断字符是否数字字符。

// 该函数将字符数字串转换成整数。输入是数字串指针的指针，返回是结果数值。另外指针将前移。
static int skip_atoi(const char **s)
{
        int i=0;

        while (is_digit(**s))
                i = i*10 + *((*s)++) - '';
        return i;
}

// 这里定义转换类型的各种符号常数。
#define ZEROPAD 1               /* pad with zero */         /* 填充零 */
#define SIGN    2               /* unsigned/signed long */ /* 无符号/符号长整数 */
#define PLUS    4               /* show plus */             /* 显示加 */
#define SPACE   8               /* space if plus */         /* 如是加，则置空格 */
#define LEFT    16              /* left justified */       /* 左调整 */
#define SPECIAL 32              /* 0x */                   /* 0x */
#define SMALL   64              /* use 'abcdef' instead of 'ABCDEF' */ /* 使用小写字母 */

// 除操作。输入：n 为被除数，base 为除数；结果：n 为商，函数返回值为余数。
// 参见 4.5.3 节有关嵌入汇编的信息。
#define do_div(n,base) ({ \
int __res; \
__asm__("divl %4":"=a" (n),"=d" (__res):"" (n),"1" (0),"r" (base)); \
__res; })


// 将整数转换为指定进制的字符串。
// 输入：num-整数；base-进制；size-字符串长度；precision-数字长度(精度)；type-类型选项。
// 输出：str 字符串指针。
static char * number(char * str, int num, int base, int size, int precision
        ,int type)
{
```

```
        char c,sign,tmp[36];
        const char *digits="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        int i;

// 如果类型 type 指出用小写字母，则定义小写字母集。
// 如果类型指出要左调整（靠左边界），则屏蔽类型中的填零标志。
// 如果进制基数小于 2 或大于 36，则退出处理，也即本程序只能处理基数在 2-32 之间的数。
        if (type&SMALL) digits="0123456789abcdefghijklmnopqrstuvwxyz";
        if (type&LEFT) type &= ~ZEROPAD;
        if (base<2 || base>36)
                return 0;
// 如果类型指出要填零，则置字符变量 c='0'（也即''），否则 c 等于空格字符。
// 如果类型指出是带符号数并且数值 num 小于 0，则置符号变量 sign=负号，并使 num 取绝对值。
// 否则如果类型指出是加号，则置 sign=加号，否则若类型带空格标志则 sign=空格，否则置 0。
        c = (type & ZEROPAD) ? '' : ' ' ;
        if (type&SIGN && num<0) {
                sign='-';
                num = -num;
        } else
                sign=(type&PLUS) ? '+' : ((type&SPACE) ? ' ' : 0);
// 若带符号，则宽度值减 1。若类型指出是特殊转换，则对于十六进制宽度再减少 2 位(用于 0x)，
// 对于八进制宽度减 1（用于八进制转换结果前放一个零）。
        if (sign) size--;
        if (type&SPECIAL)
                if (base==16) size -= 2;
                else if (base==8) size--;
// 如果数值 num 为 0，则临时字符串='0'；否则根据给定的基数将数值 num 转换成字符形式。
        i=0;
        if (num==0)
                tmp[i++]='';
        else while (num!=0)
                tmp[i++]=digits[do_div(num,base)];
// 若数值字符个数大于精度值，则精度值扩展为数字个数值。
// 宽度值 size 减去用于存放数值字符的个数。
        if (i>precision) precision=i;
        size -= precision;

// 从这里真正开始形成所需要的转换结果，并暂时放在字符串 str 中。
// 若类型中没有填零(ZEROPAD)和左靠齐（左调整）标志，则在 str 中首先
// 填放剩余宽度值指出的空格数。若需带符号位，则存入符号。
        if (!(type&(ZEROPAD+LEFT)))
                while(size-->0)
                        *str++ = ' ';
        if (sign)
                *str++ = sign;
// 若类型指出是特殊转换，则对于八进制转换结果头一位放置一个'0'；而对于十六进制则存放'0x'。
        if (type&SPECIAL)
                if (base==8)
                        *str++ = '';
                else if (base==16) {
                        *str++ = '';
                        *str++ = digits[33];    // 'X'或'x'
                }
```

```
    // 若类型中没有左调整（左靠齐）标志，则在剩余宽度中存放 c 字符（'0'或空格），见 51 行。
80          if (!(type&LEFT))
81                  while(size-->0)
82                          *str++ = c;
    // 此时 i 存有数值 num 的数字个数。若数字个数小于精度值，则 str 中放入（精度值-i）个'0'。
83          while(i<precision--)
84                  *str++ = '';
    // 将转数值换好的数字字符填入 str 中。共 i 个。
85          while(i-->0)
86                  *str++ = tmp[i];
    // 若宽度值仍大于零，则表示类型标志中有左靠齐标志标志。则在剩余宽度中放入空格。
87          while(size-->0)
88                  *str++ = ' ';
89          return str;     // 返回转换好的字符串。
90  }
91
    // 下面函数是送格式化输出到字符串中。
    // 为了能在内核中使用格式化的输出，Linus 在内核实现了该 C 标准函数。
    // 其中参数 fmt 是格式字符串；args 是个数变化的值；buf 是输出字符串缓冲区。
    // 请参见本代码列表后的有关格式转换字符的介绍。
92  int vsprintf(char *buf, const char *fmt, va_list args)
93  {
94          int len;
95          int i;
96          char * str;             // 用于存放转换过程中的字符串。
97          char *s;
98          int *ip;
99
100         int flags;              /* flags to number() */
101                                 /* number()函数使用的标志 */
102         int field_width;        /* width of output field */
                                    /* 输出字段宽度*/
103         int precision;          /* min. # of digits for integers; max
104                                    number of chars for from string */
                                    /* min. 整数数字个数；max. 字符串中字符个数 */
105         int qualifier;          /* 'h', 'l', or 'L' for integer fields */
106                                 /* 'h', 'l','L'用于整数字段 */
    // 首先将字符指针指向 buf，然后扫描格式字符串，对各个格式转换指示进行相应的处理。
107         for (str=buf ; *fmt ; ++fmt) {
    // 格式转换指示字符串均以'%'开始，这里从 fmt 格式字符串中扫描'%'，寻找格式转换字符串的开始。
    // 不是格式指示的一般字符均被依次存入 str。
108                 if (*fmt != '%') {
109                         *str++ = *fmt;
110                         continue;
111                 }
112
    // 下面取得格式指示字符串中的标志域，并将标志常量放入 flags 变量中。
113                 /* process flags */
114                 flags = 0;
115                 repeat:
116                         ++fmt;          /* this also skips first '%' */
117                         switch (*fmt) {
118                                 case '-': flags |= LEFT; goto repeat;    // 左靠齐调整。
```

```
                                case '+': flags |= PLUS; goto repeat;    // 放加号。
                                case ' ': flags |= SPACE; goto repeat;   // 放空格。
                                case '#': flags |= SPECIAL; goto repeat; // 是特殊转换。
                                case '': flags |= ZEROPAD; goto repeat;  // 要填零(即'0')。
                                }

// 取当前参数字段宽度域值，放入 field_width 变量中。如果宽度域中是数值则直接取其为宽度值。
// 如果宽度域中是字符'*'，表示下一个参数指定宽度。因此调用 va_arg 取宽度值。若此时宽度值
// 小于 0，则该负数表示其带有标志域'-'标志（左靠齐），因此还需在标志变量中添入该标志，并
// 将字段宽度值取为其绝对值。
                /* get field width */
                field_width = -1;
                if (is_digit(*fmt))
                        field_width = skip_atoi(&fmt);
                else if (*fmt == '*') {
                        /* it's the next argument */
                        field_width = va_arg(args, int);
                        if (field_width < 0) {
                                field_width = -field_width;
                                flags |= LEFT;
                        }
                }

// 下面这段代码，取格式转换串的精度域，并放入 precision 变量中。精度域开始的标志是'.'。
// 其处理过程与上面宽度域的类似。如果精度域中是数值则直接取其为精度值。如果精度域中是
// 字符'*'，表示下一个参数指定精度。因此调用 va_arg 取精度值。若此时宽度值小于 0，则
// 将字段精度值取为其绝对值。
                /* get the precision */
                precision = -1;
                if (*fmt == '.') {
                        ++fmt;
                        if (is_digit(*fmt))
                                precision = skip_atoi(&fmt);
                        else if (*fmt == '*') {
                                /* it's the next argument */
                                precision = va_arg(args, int);
                        }
                        if (precision < 0)
                                precision = 0;
                }

// 下面这段代码分析长度修饰符，并将其存入 qualifer 变量。（h,l,L 的含义参见列表后的说明）。
                /* get the conversion qualifier */
                qualifier = -1;
                if (*fmt == 'h' || *fmt == 'l' || *fmt == 'L') {
                        qualifier = *fmt;
                        ++fmt;
                }

// 下面分析转换指示符。
                switch (*fmt) {
// 如果转换指示符是'c'，则表示对应参数应是字符。此时如果标志域表明不是左靠齐，则该字段前面
// 放入宽度域值-1 个空格字符，然后再放入参数字符。如果宽度域还大于 0，则表示为左靠齐，则在
```

```
// 参数字符后面添加宽度值-1 个空格字符。
                case 'c':
                        if (!(flags & LEFT))
                                while (--field_width > 0)
                                        *str++ = ' ';
                        *str++ = (unsigned char) va_arg(args, int);
                        while (--field_width > 0)
                                *str++ = ' ';
                        break;

// 如果转换指示符是's'，则表示对应参数是字符串。首先取参数字符串的长度，若其超过了精度域值，
// 则扩展精度域=字符串长度。此时如果标志域表明不是左靠齐，则该字段前放入(宽度值-字符串长度)
// 个空格字符。然后再放入参数字符串。如果宽度域还大于 0，则表示为左靠齐，则在参数字符串后面
// 添加(宽度值-字符串长度)个空格字符。
                case 's':
                        s = va_arg(args, char *);
                        len = strlen(s);
                        if (precision < 0)
                                precision = len;
                        else if (len > precision)
                                len = precision;

                        if (!(flags & LEFT))
                                while (len < field_width--)
                                        *str++ = ' ';
                        for (i = 0; i < len; ++i)
                                *str++ = *s++;
                        while (len < field_width--)
                                *str++ = ' ';
                        break;

// 如果格式转换符是'o'，表示需将对应的参数转换成八进制数的字符串。调用 number()函数处理。
                case 'o':
                        str = number(str, va_arg(args, unsigned long), 8,
                                field_width, precision, flags);
                        break;

// 如果格式转换符是'p'，表示对应参数的一个指针类型。此时若该参数没有设置宽度域，则默认宽度
// 为 8，并且需要添零。然后调用 number()函数进行处理。
                case 'p':
                        if (field_width == -1) {
                                field_width = 8;
                                flags |= ZEROPAD;
                        }
                        str = number(str,
                                (unsigned long) va_arg(args, void *), 16,
                                field_width, precision, flags);
                        break;

// 若格式转换指示是'x'或'X'，则表示对应参数需要打印成十六进制数输出。'x'表示用小写字母表示。
                case 'x':
                        flags |= SMALL;
                case 'X':
```

```
204                             str = number(str, va_arg(args, unsigned long), 16,
205                                     field_width, precision, flags);
206                             break;
207
    // 如果格式转换字符是'd','i'或'u'，则表示对应参数是整数，'d','i'代表符号整数，因此需要加上
    // 带符号标志。'u'代表无符号整数。
208                     case 'd':
209                     case 'i':
210                             flags |= SIGN;
211                     case 'u':
212                             str = number(str, va_arg(args, unsigned long), 10,
213                                     field_width, precision, flags);
214                             break;
215
    // 若格式转换指示符是'n'，则表示要把到目前为止转换输出的字符数保存到对应参数指针指定的位置
中。
    // 首先利用 va_arg()取得该参数指针，然后将已经转换好的字符数存入该指针所指的位置。
216                     case 'n':
217                             ip = va_arg(args, int *);
218                             *ip = (str - buf);
219                             break;
220
    // 若格式转换符不是'%'，则表示格式字符串有错，直接将一个'%'写入输出串中。
    // 如果格式转换符的位置处还有字符，则也直接将该字符写入输出串中，并返回到 107 行继续处理
    // 格式字符串。否则表示已经处理到格式字符串的结尾处，则退出循环。
221                     default:
222                             if (*fmt != '%')
223                                     *str++ = '%';
224                             if (*fmt)
225                                     *str++ = *fmt;
226                             else
227                                     --fmt;
228                             break;
229                     }
230             }
231             *str = '\0';            // 最后在转换好的字符串结尾处添上 null。
232             return str-buf;         // 返回转换好的字符串长度值。
233 }
234
```

5.12.3 其它信息

5.12.3.1 vsprintf()的格式字符串

```
int vsprintf(char *buf, const char *fmt, va_list args)
```

vsprintf()函数是 printf()系列函数之一。这些函数都产生格式化的输出：接受确定输出格式的格式字符串 fmt，用格式字符串对个数变化的参数进行格式化，产生格式化的输出。

printf 直接把输出送到标准输出句柄 stdout。cprintf 把输出送到控制台。fprintf 把输出送到文件句柄。printf 前带'v'字符的(例如 vfprintf)表示参数是从 va_arg 数组的 va_list args 中接受。printf 前面带's'字符则表示把输出送到以 null 结尾的字符串 buf 中（此时用户应确保 buf 有足够的空间存放

字符串）。下面详细说明格式字符串的使用方法。

1. 格式字符串

printf 系列函数中的格式字符串用于控制函数转换方式、格式化和输出其参数。对于每个格式，必须有对应的参数，参数过多将被忽略。格式字符串中含有两类成份，一种是将被直接复制到输出中的简单字符；另一种是用于对对应参数进行格式化的转换指示字符串。

2. 格式指示字符串

格式指示串的形式如下：

%[flags][width][.prec][|h|l|L][type]

每一个转换指示串均需要以百分号(%)开始。其中

[flags]　　是可选择的标志字符序列；

[width]　　是可选择的的宽度指示符；

[.prec]　　是可选择的精度(precision)指示符；

[h|l|L] 是可选择的输入长度修饰符；

[type]　　是转换类型字符(或称为转换指示符)。

flags 控制输出对齐方式、数值符号、小数点、尾零、二进制、八进制或十六进制等，参见上面列表 27-33 行的注释。标志字符及其含义如下：

\#　　表示需要将相应参数转换为“特殊形式”。对于八进制(o)，则转换后的字符串的首位必须是一个零。对于十六进制(x 或 X)，则转换后的字符串需以'0x'或'0X'开头。对于 e,E,f,F,g 以及 G，则即使没有小数位，转换结果也将总是有一个小数点。对于 g 或 G，后拖的零也不会删除。

0　　转换结果应该是附零的。对于 d,i,o,u,x,X,e,E,f,g 和 G，转换结果的左边将用零填空而不是用空格。如果同时出现 0 和-标志，则 0 标志将被忽略。对于数值转换，如果给出了精度域，0 标志也被忽略。

-　　转换后的结果在相应字段边界内将作左调整（靠左）。（默认是作右调整--靠右）。n 转换例外，转换结果将在右面填空格。

' '　　表示带符号转换产生的一个正数结果前应该留一个空格。

+　　表示在一个符号转换结果之前总需要放置一个符号（+或-）。对于默认情况，只有负数使用负号。

width 指定了输出字符串宽度，即指定了字段的最小宽度值。如果被转换的结果要比指定的宽度小，则在其左边（或者右边，如果给出了左调整标志）需要填充空格或零（由 flags 标志确定）的个数等。除了使用数值来指定宽度域以外，也可以使用'*'来指出字段的宽度由下一个整型参数给出。当转换值宽度大于 width 指定的宽度时，在任何情况下小宽度值都不会截断结果。字段宽度会扩充以包含完整结果。

precision 是说明输出数字起码的个数。对于 d,I,o,u,x 和 X 转换，精度值指出了起码出现数字的个数。对于 e,E,f 和 F，该值指出在小数点之后出现的数字的个数。对于 g 或 G，指出最大有效数字个数。对于 s 或 S 转换，精度值说明输出字符串的最大字符数。

长度修饰指示符说明了整型数转换后的输出类型形式。下面叙述中‘整型数转换’代表 d,i,o,u,x 或 X 转换。

hh　　说明后面的整型数转换对应于一个带符号字符或无符号字符参数。

h　　说明后面的整型数转换对应于一个带符号整数或无符号短整数参数。

l 说明后面的整型数转换对应于一个长整数或无符号长整数参数。

ll 说明后面的整型数转换对应于一个长长整数或无符号长长整数参数。

L 说明 e,E,f,F,g 或 G 转换结果对应于一个长双精度参数。

type 是说明接受的输入参数类型和输出的格式。各个转换指示符的含义如下：

d,I 整数型参数将被转换为带符号整数。如果有精度(precision)的话，则给出了需要输出的最少数字个数。如果被转换的值数字个数较少，就会在其左边添零。默认的精度值是 1。

o,u,x,X 会将无符号的整数转换为无符号八进制(o)、无符号十进制(u)或者是无符号十六进制(x 或 X)表示方式输出。x 表示要使用小写字母（abcdef）来表示十六进制数，X 表示用大写字母（ABCDEF）表示十六进制数。如果存在精度域的话，说明需要输出的最少数字个数。如果被转换的值数字个数较少，就会在其左边添零。默认的精度值是 1。

e,E 这两个转换字符用于经四舍五入将参数转换成[-]d.ddde±dd 的形式。小数点之后的数字个数等于精度。如果没有精度域，就取默认值 6。如果精度是 0，则没有小数出现。E 表示用大写字母 E 来表示指数。指数部分总是用 2 位数字表示。如果数值为 0，那么指数就是 00。

f,F 这两个转换字符用于经四舍五入将参数转换成[-]ddd.ddd 的形式。小数点之后的数字个数等于精度。如果没有精度域，就取默认值 6。如果精度是 0，则没有小数出现。如果有小数点，那么后面起码会有 1 位数字。

g,G 这两个转换字符将参数转换为 f 或 e 的格式（如果是 G，则是 F 或 E 格式）。精度值指定了整数的个数。如果没有精度域，则其默认值为 6。如果精度为 0，则作为 1 来对待。如果转换时指数小于 -4 或大于等于精度，则采用 e 格式。小数部分后拖的零将被删除。仅当起码有一位小数时才会出现小数点。

c 参数将被转换成无符号字符并输出转换结果。

s 要求输入为指向字符串的指针，并且该字符串要以 null 结尾。如果有精度域，则只输出精度所要求的字符个数，并且字符串无须以 null 结尾。

p 以指针形式输出十六进制数。

n 用于把到目前为止转换输出的字符个数保存到由对应输入指针指定的位置中。不对参数进行转换。

% 输出一个百分号%，不进行转换。也即此时整个转换指示为%%。

5.12.4 与当前版本的区别

由于该文件也属于库函数，所以从 1.2 版内核开始就直接使用库中的函数了。也即删除了该文件。

5.13 printk.c 程序

5.13.1 功能描述

printk()是内核中使用的打印（显示）函数，功能与 C 标准函数库中的 print()相同。重新编写这么一个函数的原因是在内核中不能使用专用于用户模式的 fs 段寄存器，需要首先保存它。printk()函数首先使用 svprintf()对参数进行格式化处理，然后在保存了 fs 段寄存器的情况下调用 tty_write()进行信息的打印显示。

5.13.2 代码注释

列表 5.13 linux/kernel/printk.c 程序

```
/*
 *  linux/kernel/printk.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * When in kernel-mode, we cannot use printf, as fs is liable to
 * point to 'interesting' things. Make a printf with fs-saving, and
 * all is well.
 */
/*
 * 当处于内核模式时，我们不能使用 printf，因为寄存器 fs 指向其它不感兴趣的地方。
 * 自己编制一个 printf 并在使用前保存 fs，一切就解决了。
 */
#include <stdarg.h>        // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了-个
                           // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于
                           // vsprintf、vprintf、vfprintf 函数。
#include <stddef.h>        // 标准定义头文件。定义了 NULL, offsetof(TYPE, MEMBER)。

#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。

static char buf[1024];

// 下面该函数 vsprintf()在 linux/kernel/vsprintf.c 中 92 行开始。
extern int vsprintf(char * buf, const char * fmt, va_list args);

// 内核使用的显示函数。
int printk(const char *fmt, ...)
{
        va_list args;                // va_list 实际上是一个字符指针类型。
        int i;

        va_start(args, fmt);         // 参数处理开始函数。在（include/stdarg.h,13）
        i=vsprintf(buf,fmt,args);    // 使用格式串 fmt 将参数列表 args 输出到 buf 中。
                                     // 返回值 i 等于输出字符串的长度。
        va_end(args);                // 参数处理结束函数。
        __asm__("push %%fs\n\t"      // 保存 fs。
                "push %%ds\n\t"
                "pop %%fs\n\t"       // 令 fs = ds。
                "pushl %0\n\t"       // 将字符串长度压入堆栈(这三个入栈是调用参数)。
                "pushl $_buf\n\t"    // 将 buf 的地址压入堆栈。
                "pushl $0\n\t"       // 将数值 0 压入堆栈。是通道号 channel。
                "call _tty_write\n\t" // 调用 tty_write 函数。(kernel/chr_drv/tty_io.c,290)。
                "addl $8,%%esp\n\t"  // 跳过（丢弃）两个入栈参数(buf,channel)。
                "popl %0\n\t"        // 弹出字符串长度值，作为返回值。
                "pop %%fs"           // 恢复原 fs 寄存器。
                ::"r" (i):"ax","cx","dx"); // 通知编译器，寄存器 ax,cx,dx 值可能已经改变。
        return i;                    // 返回字符串长度。
}

```

5.14 panic.c 程序

5.14.1 功能描述

当内核程序出错时，则调用函数 panic()，显示错误信息并使系统进入死循环。在内核程序的许多地方，若出现严重出错时就要调用到该函数。在很多情况下，调用 panic()函数是一种简明的处理方法。这样做很好地遵循了 UNIX“尽量简明”的原则。

panic 是“惊慌，恐慌”的意思。在 Douglas Adams 的小说《Hitch hikers Guide to the Galaxy》（《银河徒步旅行者指南》）中，书中最有名的一句话就是“Don't Panic!”。该系列小说是 linux 骇客最常阅读的一类书籍。

5.14.2 代码注释

列表 5.14 linux/kernel/panic.c 程序

```
/*
 *  linux/kernel/panic.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * This function is used through-out the kernel (includeinh mm and fs)
 * to indicate a major problem.
 */
/*
 * 该函数在整个内核中使用（包括在 头文件*.h, 内存管理程序 mm 和文件系统 fs 中），
 * 用以指出主要的出错问题。
 */
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。

void sys_sync(void);    /* it's really int */ /* 实际上是整型 int (fs/buffer.c,44) */

// 该函数用来显示内核中出现的重大错误信息，并运行文件系统同步函数，然后进入死循环 -- 死机。
// 如果当前进程是任务 0 的话，还说明是交换任务出错，并且还没有运行文件系统同步函数。
volatile void panic(const char * s)
{
        printk("Kernel panic: %s\n\r",s);
        if (current == task[0])
                printk("In swapper task - not syncing\n\r");
        else
                sys_sync();
        for(;;);
}

```

5.15 本章小结

linux/kernel 目录下的 12 个代码文件给出了内核中最为重要的一些机制的实现，主要包括系统调用、进程调度、进程复制以及进程的终止处理四部分。

第 6 章 块设备驱动程序(block driver)

6.1 概述

列表 6.1 linux/kernel/blk_drv 目录

Name	Size	Last modified (GMT)	Description
Makefile	1951 bytes	1991-12-05 19:59:42	m
blk.h	3464 bytes	1991-12-05 19:58:01	m
floppy.c	11429 bytes	1991-12-07 00:00:38	m
hd.c	7807 bytes	1991-12-05 19:58:17	m
ll_rw_blk.c	3539 bytes	1991-12-04 13:41:42	m
ramdisk.c	2740 bytes	1991-12-06 03:08:06	m

6.2 Makefile 文件

6.2.1 功能描述

6.2.2 代码注释

列表 6.2 linux/kernel/blk_drv/Makefile 文件

```
#
# Makefile for the FREAX-kernel block device drivers.
#
# Note! Dependencies are done automagically by 'make dep', which also
# removes any old dependencies. DON'T put your own dependencies here
# unless it's something special (ie not a .c file).
#
# FREAX 内核块设备驱动程序的 Makefile 文件
# 注意！依赖关系是由'make dep'自动进行的，它也会自动去除原来的依赖信息。不要把你自己的
# 依赖关系信息放在这里，除非是特别文件的（也即不是一个.c 文件的信息）。
# (Linux 最初的名字叫 FREAX，后来被 ftp.funet.fi 的管理员改成 Linux 这个名字)。

AR      =gar     # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。
AS      =gas     # GNU 的汇编程序。
LD      =gld     # GNU 的连接程序。
LDFLAGS =-s -x   # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
CC      =gcc     # GNU C 语言编译器。
```

```
    # 下一行是 C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
    # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
    # 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
    # 单短小的函数代码嵌入调用程序中；-mstring-insns Linus 自己填加的优化选项，以后不再使用；
    # -nostdinc -I../include 不使用默认路径中的包含文件，而使用指定目录中的(../../include)。
14 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
15         -finline-functions -mstring-insns -nostdinc -I../../include
    # C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
    # 出设备或指定的输出文件中；-nostdinc -I../../include 同前。
16 CPP     =gcc -E -nostdinc -I../../include
17
    # 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
    # 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
    # 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
    # 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。其中$*.s（或$@）是自动目标变量，
    # $<代表第一个先决条件，这里即是符合条件*.c 的文件。
18 .c.s:
19         $(CC) $(CFLAGS) \
20         -S -o $*.s $<
   # 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。22 行是实现该规则的具体命令。
21 .s.o:
22         $(AS) -c -o $*.o $<
23 .c.o:                        # 类似上面，*.c 文件-→*.o 目标文件。不进行连接。
24         $(CC) $(CFLAGS) \
25         -c -o $*.o $<
26
27 OBJS  = ll_rw_blk.o floppy.o hd.o ramdisk.o       # 定义目标文件变量 OBJS。
28
    # 在有了先决条件 OBJS 后使用下面的命令连接成目标 blk_drv.a 库文件。
29 blk_drv.a: $(OBJS)
30         $(AR) rcs blk_drv.a $(OBJS)
31         sync
32
    # 下面的规则用于清理工作。当执行'make clean'时，就会执行 34--35 行上的命令，去除所有编译
    # 连接生成的文件。'rm'是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
33 clean:
34         rm -f core *.o *.a tmp_make
35         for i in *.c;do rm -f `basename $$i .c`.s;done
36
    # 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
    # 使用字符串编辑程序 sed 对 Makefile 文件（即是本文件）进行处理，输出为删除 Makefile
    # 文件中'### Dependencies'行后面的所有行（下面从 44 开始的行），并生成 tmp_make
    # 临时文件（38 行的作用）。然后对 kernel/blk_drv/目录下的每个 C 文件执行 gcc 预处理操作.
    # -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
    # 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
    # 文件名加上其依赖关系--该源文件中包含的所有头文件列表。把预处理结果都添加到临时
    # 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。
37 dep:
38         sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
39         (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,'`" "; \
40                 $(CPP) -M $$i;done) >> tmp_make
41         cp tmp_make Makefile
42
```

```
### Dependencies:
floppy.s floppy.o : floppy.c ../../include/linux/sched.h ../../include/linux/head.h \
  ../../include/linux/fs.h ../../include/sys/types.h ../../include/linux/mm.h \
  ../../include/signal.h ../../include/linux/kernel.h \
  ../../include/linux/fdreg.h ../../include/asm/system.h \
  ../../include/asm/io.h ../../include/asm/segment.h blk.h
hd.s hd.o : hd.c ../../include/linux/config.h ../../include/linux/sched.h \
  ../../include/linux/head.h ../../include/linux/fs.h \
  ../../include/sys/types.h ../../include/linux/mm.h ../../include/signal.h \
  ../../include/linux/kernel.h ../../include/linux/hdreg.h \
  ../../include/asm/system.h ../../include/asm/io.h \
  ../../include/asm/segment.h blk.h
ll_rw_blk.s ll_rw_blk.o : ll_rw_blk.c ../../include/errno.h ../../include/linux/sched.h \
  ../../include/linux/head.h ../../include/linux/fs.h \
  ../../include/sys/types.h ../../include/linux/mm.h ../../include/signal.h \
  ../../include/linux/kernel.h ../../include/asm/system.h blk.h
```

6.3 blk.h 文件

6.3.1 功能描述

这是有关硬盘块设备参数的头文件，因为只用于块设备，所以与块设备代码放在同一个地方。

6.3.2 代码注释

列表 6.3 linux/kernel/blk_drv/blk.h 文件

```
#ifndef _BLK_H
#define _BLK_H

#define NR_BLK_DEV      7        // 块设备的数量。
/*
 * NR_REQUEST is the number of entries in the request-queue.
 * NOTE that writes may use only the low 2/3 of these: reads
 * take precedence.
 *
 * 32 seems to be a reasonable number: enough to get some benefit
 * from the elevator-mechanism, but not so much as to lock a lot of
 * buffers when they are in the queue. 64 seems to be too many (easily
 * long pauses in reading when heavy writing/syncing is going on)
 */
/*
 * 下面定义的 NR_REQUEST 是请求队列中所包含的项数。
 * 注意，读操作仅使用这些项低端的 2/3；读操作优先处理。
 *
 * 32 项好象是一个合理的数字：已经足够从电梯算法中获得好处，
 * 但当缓冲区在队列中而锁住时又不显得是很大的数。64 就看上
 * 去太大了（当大量的写/同步操作运行时很容易引起长时间的暂停）。
 */
#define NR_REQUEST      32
```

```

/*
 * Ok, this is an expanded form so that we can use the same
 * request for paging requests when that is implemented. In
 * paging, 'bh' is NULL, and 'waiting' is used to wait for
 * read/write completion.
 */
/*
 * OK，下面是 request 结构的一个扩展形式，因而当实现以后我们就可以在分页请求中
 * 使用同样的请求。在分页处理中，'bh'是 NULL，而'waiting'则用于等待读/写的完成。
 */
// 下面是请求队列中项的结构。其中如果 dev=-1，则表示没有使用该项。
struct request {
        int dev;                /* -1 if no request */    // 使用的设备号。
        int cmd;                /* READ or WRITE */       // 命令(READ 或 WRITE)。
        int errors;                         //操作时产生的错误次数。
        unsigned long sector;               // 起始扇区。(1 块=2 扇区)
        unsigned long nr_sectors;           // 读/写扇区数。
        char * buffer;                      // 数据缓冲区。
        struct task_struct * waiting;       // 任务等待操作执行完成的地方。
        struct buffer_head * bh;            // 缓冲区头指针(include/linux/fs.h,68)。
        struct request * next;              // 指向下一请求项。
};

/*
 * This is used in the elevator algorithm: Note that
 * reads always go before writes. This is natural: reads
 * are much more time-critical than writes.
 */
/*
 * 下面的定义用于电梯算法：注意读操作总是在写操作之前进行。
 * 这是很自然的：读操作对时间的要求要比写严格得多。
 */
#define IN_ORDER(s1,s2) \
((s1)->cmd<(s2)->cmd || (s1)->cmd==(s2)->cmd && \
((s1)->dev < (s2)->dev || ((s1)->dev == (s2)->dev && \
(s1)->sector < (s2)->sector)))

// 块设备结构。
struct blk_dev_struct {
        void (*request_fn)(void);            // 请求操作的函数指针。
        struct request * current_request;    // 请求信息结构。
};

extern struct blk_dev_struct blk_dev[NR_BLK_DEV];    // 块设备数组，每种块设备占用一项。
extern struct request request[NR_REQUEST];           // 请求队列数组。
extern struct task_struct * wait_for_request;        // 等待请求的任务结构。

#ifdef MAJOR_NR           // 主设备号。

/*
 * Add entries as needed. Currently the only block devices
 * supported are hard-disks and floppies.
```

```
 */
/*
 * 需要时加入条目。目前块设备仅支持硬盘和软盘。
 */

#if (MAJOR_NR == 1)   // RAM 盘的主设备号是 1。根据这里的定义可以推理内存块主设备号也为 1。
/* ram disk */          /* RAM 盘（内存虚拟盘）*/
#define DEVICE_NAME "ramdisk"               // 设备名称 ramdisk。
#define DEVICE_REQUEST do_rd_request        // 设备请求函数 do_rd_request()。
#define DEVICE_NR(device) ((device) & 7)    // 设备号(0--7)。
#define DEVICE_ON(device)                   // 开启设备。虚拟盘无须开启和关闭。
#define DEVICE_OFF(device)                  // 关闭设备。

#elif (MAJOR_NR == 2)     // 软驱的主设备号是 2。
/* floppy */
#define DEVICE_NAME "floppy"                // 设备名称 floppy。
#define DEVICE_INTR do_floppy               // 设备中断处理程序 do_floppy()。
#define DEVICE_REQUEST do_fd_request        // 设备请求函数 do_fd_request()。
#define DEVICE_NR(device) ((device) & 3)    // 设备号（0--3）。
#define DEVICE_ON(device) floppy_on(DEVICE_NR(device))    // 开启设备函数 floppyon()。
#define DEVICE_OFF(device) floppy_off(DEVICE_NR(device)) // 关闭设备函数 floppyoff()。

#elif (MAJOR_NR == 3)     // 硬盘主设备号是 3。
/* harddisk */
#define DEVICE_NAME "harddisk"                // 硬盘名称 harddisk。
#define DEVICE_INTR do_hd                     // 设备中断处理程序 do_hd()。
#define DEVICE_REQUEST do_hd_request          // 设备请求函数 do_hd_request()。
#define DEVICE_NR(device) (MINOR(device)/5) // 设备号（0--1）。每个硬盘可以有 4 个分区。
#define DEVICE_ON(device)                     // 硬盘一直在工作，无须开启和关闭。
#define DEVICE_OFF(device)

#elif
/* unknown blk device */  /* 未知块设备 */
#error "unknown blk device"

#endif

#define CURRENT (blk_dev[MAJOR_NR].current_request) // CURRENT 为指定主设备号的当前请求结构。
#define CURRENT_DEV DEVICE_NR(CURRENT->dev)          // CURRENT_DEV 为 CURRENT 的设备号。

#ifdef DEVICE_INTR
void (*DEVICE_INTR)(void) = NULL;
#endif
static void (DEVICE_REQUEST)(void);

// 释放锁定的缓冲区。
extern inline void unlock_buffer(struct buffer_head * bh)
{
        if (!bh->b_lock)                    // 如果指定的缓冲区 bh 没有被上锁，则显示警告信息。
                printk(DEVICE_NAME ": free buffer being unlocked\n");
        bh->b_lock=0;                       // 否则将该缓冲区解锁。
        wake_up(&bh->b_wait);               // 唤醒等待该缓冲区的进程。
}
```

```

    // 结束请求。
extern inline void end_request(int uptodate)
{
        DEVICE_OFF(CURRENT->dev);                          // 关闭设备。
        if (CURRENT->bh) {
                CURRENT->bh->b_uptodate = uptodate;        // 置更新标志。
                unlock_buffer(CURRENT->bh);                // 解锁缓冲区。
        }
        if (!uptodate) {                                  // 如果更新标志为 0 则显示设备错误信息。
                printk(DEVICE_NAME " I/O error\n\r");
                printk("dev %04x, block %d\n\r",CURRENT->dev,
                        CURRENT->bh->b_blocknr);
        }
        wake_up(&CURRENT->waiting);                       // 唤醒等待该请求项的进程。
        wake_up(&wait_for_request);                       // 唤醒等待请求的进程。
        CURRENT->dev = -1;                                // 释放该请求项。
        CURRENT = CURRENT->next;                          // 从请求链表中删除该请求项。
}

    // 定义初始化请求宏。
#define INIT_REQUEST \
repeat: \
        if (!CURRENT) \                                   // 如果当前请求结构指针为 null 则返回。
                return; \
        if (MAJOR(CURRENT->dev) != MAJOR_NR) \            // 如果当前设备的主设备号不对则死机。
                panic(DEVICE_NAME ": request list destroyed"); \
        if (CURRENT->bh) { \
                if (!CURRENT->bh->b_lock) \          // 如果在进行请求操作时缓冲区没锁定则死机。
                        panic(DEVICE_NAME ": block not locked"); \
        }

#endif

#endif

```

6.3.3 其它信息

6.4 hd.c 文件

6.4.1 功能描述

6.4.2 代码注释

列表 6.4 linux/kernel/blk_drv/hd.c 程序

```
/*
 *  linux/kernel/hd.c
```

```
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7 /*
8  * This is the low-level hd interrupt support. It traverses the
9  * request-list, using interrupts to jump between functions. As
10  * all the functions are called within interrupts, we may not
11  * sleep. Special care is recommended.
12  *
13  *  modified by Drew Eckhardt to check nr of hd's from the CMOS.
14  */
  /*
   * 本程序是底层硬盘中断辅助程序。主要用于扫描请求列表，使用中断在函数之间跳转。
   * 由于所有的函数都是在中断里调用的，所以这些函数不可以睡眠。请特别注意。
   * 由 Drew Eckhardt 修改，利用 CMOS 信息检测硬盘数。
   */
15
16 #include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）可选项。
17 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
18 #include <linux/fs.h>     // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
19 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
20 #include <linux/hdreg.h>  // 硬盘参数头文件。定义访问硬盘寄存器端口，状态码，分区表等信息。
21 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
22 #include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
23 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
24
25 #define MAJOR_NR 3        // 硬盘主设备号是 3。
26 #include "blk.h"          // 块设备头文件。定义请求数据结构、块设备数据结构和宏函数等信息。
27
28 #define CMOS_READ(addr) ({ \  // 读 CMOS 参数宏函数。
29 outb_p(0x80|addr,0x70); \
30 inb_p(0x71); \
31 })
32
33 /* Max read/write errors/sector */
34 #define MAX_ERRORS      7     // 读/写一个扇区时最大出错次数。
35 #define MAX_HD          2     // 最多硬盘数。
36
37 static void recal_intr(void); // 硬盘中断程序在复位操作时会调用的重新校正函数(287 行)。
38
39 static int recalibrate = 1;   // 重新校正。
40 static int reset = 1;         // 复位。
41
42 /*
43  *  This struct defines the HD's and their types.
44  */
  /* 下面结构定义了硬盘参数及类型 */
  // 各字段分别是磁头数、每磁道扇区数、柱面数、写前预补偿柱面号、磁头着陆区柱面号、控制字节。
45 struct hd_i_struct {
46         int head,sect,cyl,wpcom,lzone,ctl;
47         };
```

```
#ifdef HD_TYPE      // 如果已经在 include/linux/config.h 中定义了 HD_TYPE…
struct hd_i_struct hd_info[] = { HD_TYPE };    // 取定义好的参数作为 hd_info[]的数据。
#define NR_HD ((sizeof (hd_info))/(sizeof (struct hd_i_struct)))    // 计算硬盘数。
#else               // 否则，都设为 0 值。
struct hd_i_struct hd_info[] = { {0,0,0,0,0,0},{0,0,0,0,0,0} };
static int NR_HD = 0;
#endif

// 定义硬盘分区结构。给出每个分区的物理起始扇区号、分区扇区总数。
// 其中 5 的倍数处的项（例如 hd[0]和 hd[5]等）代表整个硬盘中的参数。
static struct hd_struct {
        long start_sect;
        long nr_sects;
} hd[5*MAX_HD]={{0,0},};

// 读端口 port，共读 nr 字，保存在 buf 中。
#define port_read(port,buf,nr) \
__asm__("cld;rep;insw"::"d" (port),"D" (buf),"c" (nr):"cx","di")

// 写端口 port，共写 nr 字，从 buf 中取数据。
#define port_write(port,buf,nr) \
__asm__("cld;rep;outsw"::"d" (port),"S" (buf),"c" (nr):"cx","si")

extern void hd_interrupt(void);
extern void rd_load(void);

/* This may be used only once, enforced by 'static int callable' */
/* 下面该函数只在初始化时被调用一次。用静态变量 callable 作为可调用标志。*/
// 该函数的参数由初始化程序 init/main.c 的 init 子程序设置为指向 0x90080 处，此处存放着 setup.s
// 程序从 BIOS 取得的 2 个硬盘的基本参数表(32 字节)。硬盘参数表信息参见下面列表后的说明。
int sys_setup(void * BIOS)
{
        static int callable = 1;
        int i,drive;
        unsigned char cmos_disks;
        struct partition *p;
        struct buffer_head * bh;

// 初始化时 callable=1，当运行该函数时将其设置为 0，使本函数只能执行一次。
        if (!callable)
                return -1;
        callable = 0;
// 如果没有在 config.h 中定义硬盘参数，就从 0x90080 处读入。
#ifndef HD_TYPE
        for (drive=0 ; drive<2 ; drive++) {
                hd_info[drive].cyl = *(unsigned short *) BIOS;
                hd_info[drive].head = *(unsigned char *) (2+BIOS);
                hd_info[drive].wpcom = *(unsigned short *) (5+BIOS);
                hd_info[drive].ctl = *(unsigned char *) (8+BIOS);
                hd_info[drive].lzone = *(unsigned short *) (12+BIOS);
                hd_info[drive].sect = *(unsigned char *) (14+BIOS);
                BIOS += 16;        // 每个硬盘的参数表长 16 字节。
        }
```

```
// setup.s 程序在取 BIOS 中的硬盘参数表信息时，如果只有 1 个硬盘，就会将对应第 2 个硬盘的
// 16 字节全部清零。因此这里只要判断第 2 个硬盘柱面数是否为 0 就可以知道有没有第 2 个硬盘了。
        if (hd_info[1].cyl)
                NR_HD=2;          // 硬盘数量置为 2。
        else
                NR_HD=1;
#endif
// 设置每个硬盘的起始扇区号和扇区总数。其中编号 i*5 含义参见本程序后的有关说明。
        for (i=0 ; i<NR_HD ; i++) {
                hd[i*5].start_sect = 0;                        // 硬盘起始扇区号。
                hd[i*5].nr_sects = hd_info[i].head*
                                hd_info[i].sect*hd_info[i].cyl;  // 硬盘总扇区数。
        }

        /*
                We querry CMOS about hard disks : it could be that
                we have a SCSI/ESDI/etc controller that is BIOS
                compatable with ST-506, and thus showing up in our
                BIOS table, but not register compatable, and therefore
                not present in CMOS.

                Furthurmore, we will assume that our ST-506 drives
                <if any> are the primary drives in the system, and
                the ones reflected as drive 1 or 2.

                The first drive is stored in the high nibble of CMOS
                byte 0x12, the second in the low nibble.  This will be
                either a 4 bit drive type or 0xf indicating use byte 0x19
                for an 8 bit type, drive 1, 0x1a for drive 2 in CMOS.

                Needless to say, a non-zero value means we have
                an AT controller hard disk for that drive.


        */
/*
 * 我们对 CMOS 有关硬盘的信息有些怀疑：可能会出现这样的情况，我们有一块 SCSI/ESDI/等的
 * 控制器，它是以 ST-506 方式与 BIOS 兼容的，因而会出现在我们的 BIOS 参数表中，但却又不
 * 是寄存器兼容的，因此这些参数在 CMOS 中又不存在。
 * 另外，我们假设 ST-506 驱动器（如果有的话）是系统中的基本驱动器，也即以驱动器 1 或 2
 * 出现的驱动器。
 * 第 1 个驱动器参数存放在 CMOS 字节 0x12 的高半字节中，第 2 个存放在低半字节中。该 4 位字节
 * 信息可以是驱动器类型，也可能仅是 0xf。0xf 表示使用 CMOS 中 0x19 字节作为驱动器 1 的 8 位
 * 类型字节，使用 CMOS 中 0x1A 字节作为驱动器 2 的类型字节。
 * 总之，一个非零值意味着我们有一个 AT 控制器硬盘兼容的驱动器。
 */

// 这里根据上述原理来检测硬盘到底是否是 AT 控制器兼容的。
        if ((cmos_disks = CMOS_READ(0x12)) & 0xf0)
                if (cmos_disks & 0x0f)
                        NR_HD = 2;
                else
                        NR_HD = 1;
```

```
        else
                NR_HD = 0;
// 若 NR_HD=0，则两个硬盘都不是 AT 控制器兼容的，硬盘数据结构清零。
// 若 NR_HD=1，则将第 2 个硬盘的参数清零。
        for (i = NR_HD ; i < 2 ; i++) {
                hd[i*5].start_sect = 0;
                hd[i*5].nr_sects = 0;
        }
// 读取每一个硬盘上第 1 块数据（第 1 个扇区有用），获取其中的分区表信息。
// 首先利用函数 bread()读硬盘第 1 块数据(fs/buffer.c,267)，参数中的 0x300 是硬盘的主设备号
// (参见列表后的说明)。然后根据硬盘头 1 个扇区位置 0x1fe 处的两个字节是否为'55AA'来判断
// 该扇区中位于 0x1BE 开始的分区表是否有效。最后将分区表信息放入硬盘分区数据结构 hd 中。
        for (drive=0 ; drive<NR_HD ; drive++) {
                if (!(bh = bread(0x300 + drive*5,0))) {  // 0x300, 0x305 逻辑设备号。
                        printk("Unable to read partition table of drive %d\n\r",
                                drive);
                        panic("");
                }
                if (bh->b_data[510] != 0x55 || (unsigned char)
                    bh->b_data[511] != 0xAA) {      // 判断硬盘信息有效标志'55AA'。
                        printk("Bad partition table on drive %d\n\r",drive);
                        panic("");
                }
                p = 0x1BE + (void *)bh->b_data;     // 分区表位于硬盘第 1 扇区的 0x1BE 处。
                for (i=1;i<5;i++,p++) {
                        hd[i+5*drive].start_sect = p->start_sect;
                        hd[i+5*drive].nr_sects = p->nr_sects;
                }
                brelse(bh);      // 释放为存放硬盘块而申请的内存缓冲区页。
        }
        if (NR_HD)               // 如果有硬盘存在并且已读入分区表，则打印分区表正常信息。
                printk("Partition table%s ok.\n\r",(NR_HD>1)?"s":"");
        rd_load();               // 加载（创建）RAMDISK(kernel/blk_drv/ramdisk.c,71)。
        mount_root();            // 安装根文件系统(fs/super.c,242)。
        return (0);
}

//// 判断并循环等待驱动器就绪。
// 读硬盘控制器状态寄存器端口 HD_STATUS(0x1f7)，并循环检测驱动器就绪比特位和控制器忙位。
static int controller_ready(void)
{
        int retries=10000;

        while (--retries && (inb_p(HD_STATUS)&0xc0)!=0x40);
        return (retries);       // 返回等待循环的次数。
}

//// 检测硬盘执行命令后的状态。(win_表示温切斯特硬盘的缩写)
// 读取状态寄存器中的命令执行结果状态。返回 0 表示正常，1 出错。如果执行命令错，
// 则再读错误寄存器 HD_ERROR(0x1f1)。
static int win_result(void)
{
        int i=inb_p(HD_STATUS);  // 取状态信息。
```

```

        if ((i & (BUSY_STAT | READY_STAT | WRERR_STAT | SEEK_STAT | ERR_STAT))
                == (READY_STAT | SEEK_STAT))
                return(0); /* ok */
        if (i&1) i=inb(HD_ERROR);   // 若 ERR_STAT 置位，则读取错误寄存器。
        return (1);
}

//// 向硬盘控制器发送命令块（参见列表后的说明）。
// 调用参数：drive - 硬盘号(0-1)；            nsect - 读写扇区数；
//           sect  - 起始扇区；               head  - 磁头号；
//           cyl   - 柱面号；                 cmd   - 命令码；
//           *intr_addr() - 硬盘中断处理程序中将调用的处理函数。
static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
                unsigned int head,unsigned int cyl,unsigned int cmd,
                void (*intr_addr)(void))
{
        register int port asm("dx");    // port 变量对应寄存器 dx。

        if (drive>1 || head>15)         // 如果驱动器号(0,1)>1 或磁头号>15，则程序不支持。
                panic("Trying to write bad sector");
        if (!controller_ready())        // 如果等待一段时间后仍未就绪则出错，死机。
                panic("HD controller not ready");
        do_hd = intr_addr;              // do_hd 函数指针将在硬盘中断程序中被调用。
        outb_p(hd_info[drive].ctl,HD_CMD);      // 向控制寄存器(0x3f6)输出控制字节。
        port=HD_DATA;                           // 置 dx 为数据寄存器端口(0x1f0)。
        outb_p(hd_info[drive].wpcom>>2,++port); // 参数：写预补偿柱面号(需除 4)。
        outb_p(nsect,++port);                   // 参数：读/写扇区总数。
        outb_p(sect,++port);                    // 参数：起始扇区。
        outb_p(cyl,++port);                     // 参数：柱面号低 8 位。
        outb_p(cyl>>8,++port);                  // 参数：柱面号高 8 位。
        outb_p(0xA0|(drive<<4)|head,++port);    // 参数：驱动器号+磁头号。
        outb(cmd,++port);                       // 命令：硬盘控制命令。
}

//// 等待硬盘就绪。也即循环等待主状态控制器忙标志位复位。若仅有就绪或寻道结束标志
// 置位，则成功，返回 0。若经过一段时间仍为忙，则返回 1。
static int drive_busy(void)
{
        unsigned int i;

        for (i = 0; i < 10000; i++)             // 循环等待就绪标志位置位。
                if (READY_STAT == (inb_p(HD_STATUS) & (BUSY_STAT|READY_STAT)))
                        break;
        i = inb(HD_STATUS);                     // 再取主控制器状态字节。
        i &= BUSY_STAT | READY_STAT | SEEK_STAT;  // 检测忙位、就绪位和寻道结束位。
        if (i == READY_STAT | SEEK_STAT)        // 若仅有就绪或寻道结束标志，则返回 0。
                return(0);
        printk("HD controller times out\n\r");  // 否则等待超时，显示信息。并返回 1。
        return(1);
}

//// 诊断复位（重新校正）硬盘控制器。
```

```
static void reset_controller(void)
{
        int     i;

        outb(4,HD_CMD);                         // 向控制寄存器端口发送控制字节(4-复位)。
        for(i = 0; i < 100; i++) nop();         // 等待一段时间（循环空操作）。
        outb(hd_info[0].ctl & 0x0f ,HD_CMD);    // 再发送正常的控制字节(不禁止重试、重读)。
        if (drive_busy())                       // 若等待硬盘就绪超时，则显示出错信息。
                printk("HD-controller still busy\n\r");
        if ((i = inb(HD_ERROR)) != 1)           // 取错误寄存器，若不等于 1（无错误）则出错。
                printk("HD-controller reset failed: %02x\n\r",i);
}

//// 复位硬盘 nr。首先复位（重新校正）硬盘控制器。然后发送硬盘控制器命令“建立驱动器参数”，
// 其中 recal_intr()是在硬盘中断处理程序中调用的重新校正处理函数。
static void reset_hd(int nr)
{
        reset_controller();
        hd_out(nr,hd_info[nr].sect,hd_info[nr].sect,hd_info[nr].head-1,
                hd_info[nr].cyl,WIN_SPECIFY,&recal_intr);
}

//// 意外硬盘中断调用函数。
//发生意外硬盘中断时，硬盘中断处理程序中调用的默认 C 处理函数。在被调用函数指针为空时
// 调用该函数。参见(kernel/system_call.s,241 行)。
void unexpected_hd_interrupt(void)
{
        printk("Unexpected HD interrupt\n\r");
}

//// 读写硬盘失败处理调用函数。
static void bad_rw_intr(void)
{
        if (++CURRENT->errors >= MAX_ERRORS)    // 如果读扇区时的出错次数大于或等于 7 次时，
                end_request(0);                 // 则结束请求并唤醒等待该请求的进程，而且
                                                // 对应缓冲区更新标志复位（没有更新）。
        if (CURRENT->errors > MAX_ERRORS/2)     // 如果读一扇区时的出错次数已经大于 3 次，
                reset = 1;                      // 则要求执行复位硬盘控制器操作。
}

//// 读操作中断调用函数。将在执行硬盘中断处理程序中被调用。
static void read_intr(void)
{
        if (win_result()) {                      // 若控制器忙、读写错或命令执行错，
                bad_rw_intr();                   // 则进行读写硬盘失败处理
                do_hd_request();                 // 然后再次请求硬盘作相应(复位)处理。
                return;
        }
        port_read(HD_DATA,CURRENT->buffer,256); // 将数据从数据寄存器端口读到请求缓冲区。
        CURRENT->errors = 0;                     // 清出错次数。
        CURRENT->buffer += 512;                  // 调整缓冲区指针，指向新的空区。
        CURRENT->sector++;                       // 起始扇区号加 1，
        if (--CURRENT->nr_sectors) {             // 如果所需读出的扇区数还没有读完，则
```

```
                do_hd = &read_intr;             // 再次置硬盘调用 C 函数指针为 read_intr()
                return;                         // 因为硬盘中断处理程序每次调用 do_hd 时
        }                                       // 都会将该函数指针置空。参见 system_call.s
        end_request(1);                 // 若全部扇区数据已经读完，则处理请求结束事宜，
        do_hd_request();                // 执行其它硬盘请求操作。
}

//// 写扇区中断调用函数。在硬盘中断处理程序中被调用。
// 在写命令执行后，会产生硬盘中断信号，执行硬盘中断处理程序，此时在硬盘中断处理程序中调用的
// C 函数指针 do_hd()已经指向 write_intr()，因此会在写操作完成（或出错）后，执行该函数。
static void write_intr(void)
{
        if (win_result()) {             // 如果硬盘控制器返回错误信息，
                bad_rw_intr();          // 则首先进行硬盘读写失败处理，
                do_hd_request();        // 然后再次请求硬盘作相应(复位)处理，
                return;                 // 然后返回（也退出了此次硬盘中断）。
        }
        if (--CURRENT->nr_sectors) {    // 否则将欲写扇区数减 1，若还有扇区要写，则
                CURRENT->sector++;      // 当前请求起始扇区号+1，
                CURRENT->buffer += 512; // 调整请求缓冲区指针，
                do_hd = &write_intr;    // 置硬盘中断程序调用函数指针为 write_intr()，
                port_write(HD_DATA,CURRENT->buffer,256);  // 再向数据寄存器端口写 256 字节。
                return;                 // 返回等待硬盘再次完成写操作后的中断处理。
        }
        end_request(1);                 // 若全部扇区数据已经写完，则处理请求结束事宜，
        do_hd_request();                // 执行其它硬盘请求操作。
}

//// 硬盘重新校正（复位）中断调用函数。在硬盘中断处理程序中被调用。
// 如果硬盘控制器返回错误信息，则首先进行硬盘读写失败处理，然后请求硬盘作相应(复位)处理。
static void recal_intr(void)
{
        if (win_result())
                bad_rw_intr();
        do_hd_request();
}

// 执行硬盘读写请求操作。
void do_hd_request(void)
{
        int i,r;
        unsigned int block,dev;
        unsigned int sec,head,cyl;
        unsigned int nsect;

        INIT_REQUEST;                   // 检测请求项的合法性(参见 kernel/blk_drv/blk.h,127)。
        dev = MINOR(CURRENT->dev);      // 取设备号中的子设备号(见列表后对硬盘设备号的说明)。
                                        // CURRENT 定义为(blk_dev[MAJOR_NR].current_request)。
        block = CURRENT->sector;        // 请求的起始扇区。
// 如果子设备号不存在或者起始扇区大于该分区扇区数-2，则结束该请求，并跳转到标号 repeat 处
// （定义在 INIT_REQUEST 开始处）。
        if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
                end_request(0);
```

```
306                 goto repeat;          // 该标号在 blk.h 最后面。
307         }
308         block += hd[dev].start_sect; // 将所需读的块对应到硬盘绝对扇区号。
309         dev /= 5;                    // 此时 dev 代表硬盘号（0 或 1）。
// 下面嵌入汇编代码用来从硬盘信息结构中根据起始扇区号和每磁道扇区数计算在磁道中的
// 扇区号(sec)、所在柱面号(cyl)和磁头号(head)
310         __asm__("divl %4":"=a" (block),"=d" (sec):"" (block),"1" (0),
311                 "r" (hd_info[dev].sect));
312         __asm__("divl %4":"=a" (cyl),"=d" (head):"" (block),"1" (0),
313                 "r" (hd_info[dev].head));
314         sec++;
315         nsect = CURRENT->nr_sectors;  // 欲读/写的扇区数。
// 如果 reset 置 1，则执行复位操作。复位硬盘和控制器，并置需要重新校正标志，返回。
316         if (reset) {
317                 reset = 0;
318                 recalibrate = 1;
319                 reset_hd(CURRENT_DEV);
320                 return;
321         }
// 如果重新校正标志(recalibrate)置位，则首先复位该标志，然后向硬盘控制器发送重新校正命令。
322         if (recalibrate) {
323                 recalibrate = 0;
324                 hd_out(dev,hd_info[CURRENT_DEV].sect,0,0,0,
325                         WIN_RESTORE,&recal_intr);
326                 return;
327         }
// 如果当前请求是写扇区操作，则发送写命令，循环读取状态寄存器信息并判断请求服务标志
// DRQ_STAT 是否置位。
328         if (CURRENT->cmd == WRITE) {
329                 hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);
330                 for(i=0 ; i<3000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
331                         /* nothing */ ;
// 如置位则退出循环。若等到循环结束 DRQ_STAT 也没有置位，则此次写硬盘操作失败，去处理下一个
// 硬盘请求。否则向硬盘控制器数据寄存器端口 HD_DATA 写入 1 个扇区的数据。
332                 if (!r) {
333                         bad_rw_intr();
334                         goto repeat;    // 该标号在 blk.h 最后面，也即跳到 301 行。
335                 }
336                 port_write(HD_DATA,CURRENT->buffer,256);
// 如果当前请求是读硬盘扇区，则向硬盘控制器发送读扇区命令。
337         } else if (CURRENT->cmd == READ) {
338                 hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
339         } else
340                 panic("unknown hd-command");
341 }
342
// 硬盘系统初始化。
343 void hd_init(void)
344 {
345         blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;  // do_hd_request()。
346         set_intr_gate(0x2E,&hd_interrupt);  //设置硬盘中断门向量 int 0x2E(46)。
                                            // hd_interrupt 在(kernel/system_call.s,221)。
347         outb_p(inb_p(0x21)&0xfb,0x21);      // 复位接联的主 8259A int2 的屏蔽位，允许从片
```

```
                                          // 发出中断请求信号。
          outb(inb_p(0xA1)&0xbf,0xA1);    // 复位硬盘的中断请求屏蔽位（在从片上），允许
                                          // 硬盘控制器发送中断请求信号。
}

```

6.4.3 其它信息

6.4.3.1 AT 硬盘接口寄存器

AT 硬盘控制器的编程寄存器端口见下表所示。

表 6.1 AT 硬盘控制器寄存器端口及作用

I/O 端口	读操作	写操作
0x1f0	数据寄存器　　　　-- 扇区数据（读、写、格式化）	
0x1f1	错误寄存器（错误状态）	写前预补偿寄存器
0x1f2	扇区数寄存器　　-- 扇区数（读、写、检验、格式化）	
0x1f3	扇区号寄存器　　　-- 起始扇区（读、写、检验）	
0x1f4	柱面号寄存器　　　-- 柱面号低字节（读、写、检验、格式化）	
0x1f5	柱面号寄存器　　　-- 柱面号高字节（读、写、检验、格式化）	
0x1f6	驱动器/磁头寄存器 -- 驱动器号/磁头号（101dhhhh, d=驱动器号，h=磁头号）	
0x1f7	主状态寄存器	命令寄存器
0x3f6	---	硬盘控制寄存器
0x3f7	数字输入寄存器（与 1.2M 软盘合用）	---

下面对各端口寄存器进行详细说明。

1. 数据寄存器（HD_DATA，0x1f0）

这是一对 16 位高速 PIO 数据传输器，用于扇区读、写和磁道格式化操作。CPU 通过该数据寄存器向硬盘写入或从硬盘读出 1 个扇区的数据，也即要使用命令'rep outsw'或'rep insw'重复读/写 cx=256 字。

2. 错误寄存器（读）/写前预补偿寄存器（写）（HD_ERROR，0x1f1）

在读时，该寄存器存放有 8 位的错误状态。但只有当主状态寄存器(HD_STATUS，0x1f7)的位 0=1 时该寄存器中的数据才有效。执行控制器诊断命令时的含义与其它命令时的不同。下面分别列出：

表 6.2 硬盘控制器错误寄存器

值	诊断命令时	其它命令时
0x01	无错误	数据标志丢失
0x02	控制器出错	磁道 0 错
0x03	扇区缓冲区错	
0x04	ECC 部件错	命令放弃
0x05	控制处理器错	
0x10		ID 未找到
0x40		ECC 错误
0x80		坏扇区

在写操作时，该寄存器即作为写前预补偿寄存器。它记录写预补偿起始柱面号。对应与硬盘基本参

数表位移 0x05 处的一个字，需除 4 后输出。

3. 扇区数寄存器（HD_NSECTOR，0x1f2）

该寄存器存放读、写、检验和格式化命令指定的扇区数。当用于多扇区操作时，每完成 1 扇区的操作该寄存器就自动减 1，直到为 0。若初值为 0，则表示传输最大扇区数 256。

4. 扇区号寄存器（HD_SECTOR，0x1f3）

该寄存器存放读、写、检验操作命令指定的扇区号。在多扇区操作时，保存的是起始扇区号，而每完成 1 扇区的操作就自动增 1。

5. 柱面号寄存器（HD_LCYL，HD_HCYL，0x1f4，0x1f5）

该两个柱面号寄存器分别存放有柱面号的低 8 位和高 2 位。

6. 驱动器/磁头寄存器(HD_CURRENT，0x1f6)

该寄存器存放有读、写、检验、寻道和格式化命令指定的驱动器和磁头号。其位格式为 101dhhhh。其中 101 表示采用 ECC 校验码和每扇区为 512 字节；d 表示选择的驱动器（0 或 1)；hhhh 表示选择的磁头。

7. 主状态寄存器（读）/命令寄存器（写）(HD_STATUS/HD_COMMAND，0x1f7)

在读时，对应一个 8 位主状态寄存器。反映硬盘控制器在执行命令前后的操作状态。各位的含义如下：

```
ERR_STAT     0x01   // 命令执行错误。
INDEX_STAT   0x02   // 收到索引。
ECC_STAT     0x04   // ECC 校验错。
DRQ_STAT     0x08   // 请求服务。
SEEK_STAT    0x10   // 寻道结束。
WRERR_STAT   0x20   // 驱动器故障。
READY_STAT   0x40   // 驱动器准备好（就绪)。
BUSY_STAT    0x80   // 控制器忙碌。
```

当写时，该端口对应命令寄存器，接受 CPU 发出的硬盘控制命令，共有 8 种命令，见下表所示。

表 6.3 AT 硬盘控制器命令列表

命令名称	命令码字节		命令执行结束形式
	高 4 位	D3 D2 D1 D0	
驱动器重新校正(复位)	0x1	R R R R	中断
读扇区	0x2	0 0 L T	中断
写扇区	0x3	0 0 L T	中断
扇区检验	0x4	0 0 0 T	中断
格式化磁道	0x5	0 0 0 0	中断
控制器初始化	0x6	0 0 0 0	中断
寻道	0x7	R R R R	中断
控制器诊断	0x9	0 0 0 0	控制器空闲
建立驱动器参数	0x9	0 0 0 1	控制器空闲

表中命令码字节的低 4 位是附加参数，其含义为：

R 是步进速率。R=0，则步进速率为 35us；R=1 为 0.5ms，以此量递增。

L 是数据模式。L=0 表示读/写扇区为 512 字节；L=1 表示读/写扇区为 512 加 4 字节的 ECC 码。

T 是重试模式。T=0 表示允许重试；T=1 则禁止重试。

8. 硬盘控制寄存器（写）（HD_CMD，0x3f6）

该寄存器是只写的。用于存放硬盘控制字节并控制复位操作。其定义参见下面硬盘基本参数表的位移 0x08 处的字节说明。

6.4.3.2 AT 硬盘控制器编程

在对硬盘控制器进行操作控制时，需要同时发送参数和命令。其命令格式如下表所示。首先发送 6 字节的参数，最后发出 1 字节的命令码。不管什么命令均需要完整输出这 7 字节的命令块，依次写入端口 0x1f1 -- 0x1f7。。

表 6.4 命令格式

0x1f1	写预补偿起始柱面号
0x1f2	扇区数
0x1f3	起始扇区号
0x1f4	柱面号低字节
0x1f5	柱面号高字节
0x1f6	驱动器号/磁头号
0x1f7	命令码

首先 CPU 向控制寄存器端口(HD_CMD)0x3f6 输出控制字节，建立相应的硬盘控制方式。方式建立后即可按上面顺序发送参数和命令。步骤为：

1. 检测控制器空闲状态：CPU 通过读主状态寄存器，若位 7 为 0，表示控制器空闲。若在规定时间内控制器一直处于忙状态，则判为超时出错。
2. 检测驱动器就绪：CPU 判断主状态寄存器位 6 是否为 1 来看驱动器是否就绪。为 1 则可输出参数和命令。
3. 输出命令块：按顺序输出分别向对应端口输出参数和命令。
4. CPU 等待中断产生：命令执行后，由硬盘控制器产生中断请求信号（IRQ14 -- 对应中断 int46）或置控制器状态为空闲，表明操作结束或表示请求扇区传输（多扇区读/写）。
5. 检测操作结果：CPU 再次读主状态寄存器，若位 0 等于 0 则表示命令执行成功，否则失败。若失败则可进一步查询错误寄存器(HD_ERROR)取错误码。

6.4.3.3 硬盘基本参数表

中断向量表中，int 0x41 的中断向量位置（4 * 0x41 =0x0000:0x0104）存放的并不是中断程序的地址而是第一个硬盘的基本参数表。对于 100%兼容的 BIOS 来说，这里存放着硬盘参数表阵列的首地址 F000h:E401h。第二个硬盘的基本参数表入口地址存于 int 0x46 中断向量中。

表 6.5 硬盘基本参数信息表

位移	大小	说明
0x00	字	柱面数
0x02	字节	磁头数
0x03	字	开始减小写电流的柱面(仅 PC XT 使用，其它为 0)
0x05	字	开始写前预补偿柱面号（乘 4）
0x07	字节	最大 ECC 猝发长度（仅 XT 使用，其它为 0）

0x08	字节	控制字节（驱动器步进选择） 位 0　　未用 位 1　　保留(0) (关闭 IRQ) 位 2　　允许复位 位 3　　若磁头数大于 8 则置 1 位 4　　未用(0) 位 5　　若在柱面数+1 处有生产商的坏区图，则置 1 位 6　　禁止 ECC 重试 位 7　　禁止访问重试。
0x09	字节	标准超时值（仅 XT 使用，其它为 0）
0x0A	字节	格式化超时值（仅 XT 使用，其它为 0）
0x0B	字节	检测驱动器超时值（仅 XT 使用，其它为 0）
0x0C	字	磁头着陆(停止)柱面号
0x0E	字节	每磁道扇区数
0x0F	字节	保留。

6.4.3.4 硬盘设备号命名方式

硬盘的主设备号是 3。其它设备的主设备号分别为：

1-内存,2-磁盘,3-硬盘,4-ttyx,5-tty,6-并行口,7-非命名管道

由于 1 个硬盘中可以存在 1--4 个分区，因此硬盘还依据分区的不同用次设备号进行指定分区。因此硬盘的逻辑设备号由以下方式构成：

设备号=主设备号*256 + 次设备号

也即 dev_no = (major<<8) + minor

两个硬盘的所有逻辑设备号见下表所示。

表 6.6 硬盘逻辑设备号

逻辑设备号	对应设备文件	说明
0x300	/dev/hd0	代表整个第 1 个硬盘
0x301	/dev/hd1	表示第 1 个硬盘的第 1 个分区
0x302	/dev/hd2	表示第 1 个硬盘的第 2 个分区
0x303	/dev/hd3	表示第 1 个硬盘的第 3 个分区
0x304	/dev/hd4	表示第 1 个硬盘的第 4 个分区
0x305	/dev/hd5	代表整个第 2 个硬盘
0x306	/dev/hd6	表示第 2 个硬盘的第 1 个分区
0x307	/dev/hd7	表示第 2 个硬盘的第 2 个分区
0x308	/dev/hd8	表示第 2 个硬盘的第 3 个分区
0x309	/dev/hd9	表示第 2 个硬盘的第 4 个分区

其中 0x300 和 0x305 并不与哪个分区对应，而是代表整个硬盘。

从 linux 内核 0.95 版后已经不使用这种烦琐的命名方式，而是使用与现在相同的命名方法了。

6.4.3.5 硬盘分区表

为了实现多个操作系统共享硬盘资源，硬盘可以在逻辑上分为 1--4 个分区。每个分区之间的扇区号是邻接的。分区表由 4 个表项组成，每个表项由 16 字节组成，对应一个分区的信息，存放有分区的大小和起止的柱面号、磁道号和扇区号，见下表所示。分区表存放在硬盘的 0 柱面 0 头第 1 个扇区的

0x1BE--0x1FD 处。

表 6.7 硬盘分区表结构

位置	名称	大小	说明
0x00	boot_ind	字节	引导标志。4 个分区中同时只能有一个分区是可引导的。0x00-不从该分区引导操作系统；0x80-从该分区引导操作系统。
0x01	head	字节	分区起始磁头号。
0x02	sector	字节	分区起始扇区号(位 0-5)和起始柱面号高 2 位(位 6-7)。
0x03	cyl	字节	分区起始柱面号低 8 位。
0x04	sys_ind	字节	分区类型字节。0x0b-DOS；0x80-Old Minix；0x83-Linux …
0x05	end_head	字节	分区的结束磁头号。
0x06	end_sector	字节	结束扇区号(位 0-5)和结束柱面号高 2 位(位 6-7)。
0x07	end_cyl	字节	结束柱面号低 8 位。
0x08--0x0b	start_sect	长字	分区起始物理扇区号。
0x0c--0x0f	nr_sects	长字	分区占用的扇区数。

6.5 ll_rw_blk.c 文件

6.5.1 功能描述

该程序主要用于执行低层块设备读/写操作。与上层其它程序的接口是 ll_rw_block()函数。调用该函数会引发所有的数据块请求操作被完成。因为该函数会调用到函数 do_hd_request()，以执行硬盘数据读/写操作，而 do_hd_request()也是硬盘中断处理过程中调用的 C 函数。因此，当硬盘操作完成或出错发出硬盘中断(int 0x2e)时，又会在中断处理中再次调用 do_hd_request()，若在硬盘操作期间又有新的请求加入，则 do_hd_request 会再次执行硬盘读/写操作，形成循环调用，直到所有的数据读/写请求操作都完成。见图 6.x 所示。

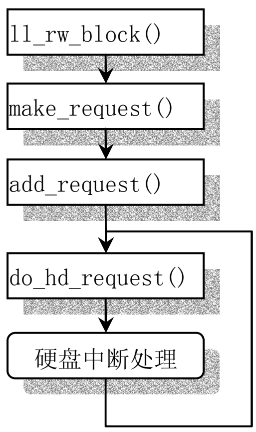


图 5.x ll_rw_block 调用序列

6.5.2 代码注释

列表 6.5 linux/kernel/blk_drv/ll_rw_blk.c 程序

```
/*
 *  linux/kernel/blk_dev/ll_rw.c
 *
 * (C) 1991 Linus Torvalds
 */

/*
 * This handles all read/write requests to block devices
 */
/*
 * 该程序处理块设备的所有读/写操作。
 */
#include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)
#include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                            // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/system.h>     // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

#include "blk.h"            // 块设备头文件。定义请求数据结构、块设备数据结构和宏函数等信息。

/*
 * The request-struct contains all necessary data
 * to load a nr of sectors into memory
 */
/*
 * 请求结构中含有加载 nr 扇区数据到内存的所有必须的信息。
 */
struct request request[NR_REQUEST];

/*
 * used to wait on when there are no free requests
 */
/* 是用于请求数组没有空闲项时的临时等待处 */
struct task_struct * wait_for_request = NULL;

/* blk_dev_struct is:
 *       do_request-address
 *       next-request
 */
/* blk_dev_struct 块设备结构是：(kernel/blk_drv/blk.h,23)
 *       do_request-address     //对应主设备号的请求处理程序指针。
 *       current-request           // 该设备的下一个请求。
 */
// 该数组使用主设备号作为索引（下标）。
struct blk_dev_struct blk_dev[NR_BLK_DEV] = {
        { NULL, NULL },          /* no_dev */     // 0 - 无设备。
        { NULL, NULL },          /* dev mem */    // 1 - 内存。
        { NULL, NULL },          /* dev fd */     // 2 - 软驱设备。
        { NULL, NULL },          /* dev hd */     // 3 - 硬盘设备。
        { NULL, NULL },          /* dev ttyx */   // 4 - ttyx 设备。
```

```
        { NULL, NULL },          /* dev tty */   // 5 - tty 设备。
        { NULL, NULL }           /* dev lp */    // 6 - lp 打印机设备。
};

// 锁定指定的缓冲区 bh。如果指定的缓冲区已经被其它任务锁定，则使自己睡眠（不可中断地等待），
// 直到被执行解锁缓冲区的任务明确地唤醒。
static inline void lock_buffer(struct buffer_head * bh)
{
        cli();                          // 清中断许可。
        while (bh->b_lock)              // 如果缓冲区已被锁定，则睡眠，直到缓冲区解锁。
                sleep_on(&bh->b_wait);
        bh->b_lock=1;                   // 立刻锁定该缓冲区。
        sti();                          // 开中断。
}

// 释放（解锁）锁定的缓冲区。
static inline void unlock_buffer(struct buffer_head * bh)
{
        if (!bh->b_lock)               // 如果该缓冲区并没有被锁定，则打印出错信息。
                printk("ll_rw_block.c: buffer not locked\n\r");
        bh->b_lock = 0;                // 清锁定标志。
        wake_up(&bh->b_wait);          // 唤醒等待该缓冲区的任务。
}

/*
 * add-request adds a request to the linked list.
 * It disables interrupts so that it can muck with the
 * request-lists in peace.
 */
/*
 * add-request()向连表中加入一项请求。它关闭中断，
 * 这样就能安全地处理请求连表了 */
 */
//// 向链表中加入请求项。参数 dev 指定块设备，req 是请求的结构信息。

static void add_request(struct blk_dev_struct * dev, struct request * req)
{
        struct request * tmp;

        req->next = NULL;
        cli();                                    // 关中断。
        if (req->bh)
                req->bh->b_dirt = 0;              // 清缓冲区“脏”标志。
// 如果 dev 的当前请求(current_request)子段为空，则表示目前该设备没有请求项，本次是第 1 个
// 请求项，因此可将块设备当前请求指针直接指向请求项，并立刻执行相应设备的请求函数。
        if (!(tmp = dev->current_request)) {
                dev->current_request = req;
                sti();                 // 开中断。
                (dev->request_fn)();   // 执行设备请求函数，对于硬盘(3)是 do_hd_request()。
                return;
        }
// 如果目前该设备已经有请求项在等待，则首先利用电梯算法搜索最佳位置，然后将当前请求插入
// 请求链表中。
```

```
        for ( ; tmp->next ; tmp=tmp->next)
                if ((IN_ORDER(tmp,req) ||
                    !IN_ORDER(tmp,tmp->next)) &&
                    IN_ORDER(req,tmp->next))
                        break;
        req->next=tmp->next;
        tmp->next=req;
        sti();
}

//// 创建请求项并插入请求队列。参数是：主设备号 major，命令 rw，存放数据的缓冲区头指针 bh。
static void make_request(int major,int rw, struct buffer_head * bh)
{
        struct request * req;
        int rw_ahead;

/* WRITEA/READA is special case - it is not really needed, so if the */
/* buffer is locked, we just forget about it, else it's a normal read */
/* WRITEA/READA 是特殊的情况 - 它们并不是必要的，所以如果缓冲区已经上锁，*/
/* 我们就不管它而退出，否则的话就执行一般的读/写操作。 */
// 这里'READ'和'WRITE'后面的'A'字符代表英文单词 Ahead，表示提前预读/写数据块的意思。
// 当指定的缓冲区正在使用，已被上锁时，就放弃预读/写请求。
        if (rw_ahead = (rw == READA || rw == WRITEA)) {
                if (bh->b_lock)
                        return;
                if (rw == READA)
                        rw = READ;
                else
                        rw = WRITE;
        }
// 如果命令不是 READ 或 WRITE 则表示内核程序有错，显示出错信息并死机。
        if (rw!=READ && rw!=WRITE)
                panic("Bad block dev command, must be R/W/RA/WA");
// 锁定缓冲区，如果缓冲区已经上锁，则当前任务（进程）就会睡眠，直到被明确地唤醒。
        lock_buffer(bh);
// 如果命令是写并且缓冲区数据不脏，或者命令是读并且缓冲区数据是更新过的，则不用添加
// 这个请求。将缓冲区解锁并退出。
        if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
                unlock_buffer(bh);
                return;
        }
repeat:
/* we don't allow the write-requests to fill up the queue completely:
 * we want some room for reads: they take precedence. The last third
 * of the requests are only for reads.
 */
/* 我们不能让队列中全都是写请求项：我们需要为读请求保留一些空间：读操作
 * 是优先的。请求队列的后三分之一空间是为读准备的。
 */
// 请求项是从请求数组末尾开始搜索空项填入的。根据上述要求，对于读命令请求，可以直接
// 从队列末尾开始操作，而写请求则只能从队列的 2/3 处向头上搜索空项填入。
        if (rw == READ)
                req = request+NR_REQUEST;          // 对于读请求，将队列指针指向队列尾部。
```

```
        else
                req = request+((NR_REQUEST*2)/3); // 对于写请求，队列指针指向队列 2/3 处。
/* find an empty request */
/* 搜索一个空请求项 */
// 从后向前搜索，当请求结构 request 的 dev 字段值=-1 时，表示该项未被占用。
        while (--req >= request)
                if (req->dev<0)
                        break;
/* if none found, sleep on new requests: check for rw_ahead */
/* 如果没有找到空闲项，则让该次新请求睡眠：需检查是否提前读/写 */
// 如果没有一项是空闲的（此时 request 数组指针已经搜索越过头部），则查看此次请求是否是
// 提前读/写（READA 或 WRITEA），如果是则放弃此次请求。否则让本次请求睡眠（等待请求队列
// 腾出空项），过一会再来搜索请求队列。
        if (req < request) {                    // 如果请求队列中没有空项，则
                if (rw_ahead) {                 // 如果是提前读/写请求，则解锁缓冲区，退出。
                        unlock_buffer(bh);
                        return;
                }
                sleep_on(&wait_for_request); // 否则让本次请求睡眠，过会再查看请求队列。
                goto repeat;
        }
/* fill up the request-info, and add it to the queue */
/* 向空闲请求项中填写请求信息，并将其加入队列中 */
// 请求结构参见（kernel/blk_drv/blk.h,23）。
        req->dev = bh->b_dev;                 // 设备号。
        req->cmd = rw;                        // 命令(READ/WRITE)。
        req->errors=0;                        // 操作时产生的错误次数。
        req->sector = bh->b_blocknr<<1;       // 起始扇区。(1 块=2 扇区)
        req->nr_sectors = 2;                  // 读写扇区数。
        req->buffer = bh->b_data;             // 数据缓冲区。
        req->waiting = NULL;                  // 任务等待操作执行完成的地方。
        req->bh = bh;                         // 缓冲区头指针。
        req->next = NULL;                     // 指向下一请求项。
        add_request(major+blk_dev,req);       // 将请求项加入队列中(blk_dev[major],req)。
}

//// 低层读写数据块函数。
// 该函数主要是在 fs/buffer.c 中被调用。实际的读写操作是由设备的 request_fn()函数完成。
// 对于硬盘操作，该函数是 do_hd_request()。（kernel/blk_drv/hd.c,294）
void ll_rw_block(int rw, struct buffer_head * bh)
{
        unsigned int major;           // 主设备号（对于硬盘是 3）。

// 如果设备的主设备号不存在或者该设备的读写操作函数不存在，则显示出错信息，并返回。
        if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
        !(blk_dev[major].request_fn)) {
                printk("Trying to read nonexistent block-device\n\r");
                return;
        }
        make_request(major,rw,bh);  //  创建请求项并插入请求队列。
}

//// 块设备初始化函数，由初始化程序 main.c 调用（init/main.c,128）。
```

```
// 初始化请求数组，将所有请求项置为空闲项(dev = -1)。有 32 项(NR_REQUEST = 32)。
void blk_dev_init(void)
{
        int i;

        for (i=0 ; i<NR_REQUEST ; i++) {
                request[i].dev = -1;
                request[i].next = NULL;
        }
}

```

6.6 ramdisk.c 文件

6.6.1 功能描述

6.6.2 代码注释

列表 6.6 linux/kernel/blk_drv/ramdisk.c 程序

```
/*
 *  linux/kernel/blk_drv/ramdisk.c
 *
 *  Written by Theodore Ts'o, 12/2/91
 */
/* 由 Theodore Ts'o 编制，12/2/91
 */
// Theodore Ts'o (Ted Ts'o)是 linux 社区中的著名人物。Linux 在世界范围内的流行也有他很大的
// 功劳，早在 Linux 操作系统刚问世时，他就怀着极大的热情为 linux 的发展提供了 maillist，并
// 在北美洲地区最早设立了 linux 的 ftp 站点（tsx-11.mit.edu），而且至今仍然为广大 linux 用户
// 提供服务。他对 linux 作出的最大贡献之一是提出并实现了 ext2 文件系统。该文件系统已成为
// linux 世界中事实上的文件系统标准。最近他又推出了 ext3 文件系统，大大提高了文件系统的
// 稳定性和访问效率。作为对他的推崇，第 97 期（2002 年 5 月）的 linuxjournal 期刊将他作为
// 了封面人物，并对他进行了采访。目前，他为 IBM linux 技术中心工作，并从事着有关 LSB
// (Linux Standard Base)等方面的工作。(他的主页：http://thunk.org/tytso/)

#include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。

#include <linux/config.h>  // 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）可选项。
#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/fs.h>      // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
#include <linux/kernel.h>  // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/system.h>    // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
#include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
#include <asm/memory.h>    // 内存拷贝头文件。含有 memcpy()嵌入式汇编宏函数。

#define MAJOR_NR 1         // 内存主设备号是 1。
```

```
18 #include "blk.h"
19
20 char    *rd_start;          // 虚拟盘在内存中的起始位置。在 52 行初始化函数 rd_init()中
                               // 确定。参见(init/main.c,124)（缩写 rd_代表 ramdisk_）。
21 int     rd_length = 0;      // 虚拟盘所占内存大小（字节）。
22
   // 执行虚拟盘(ramdisk)读写操作。程序结构与 do_hd_request()类似(kernel/blk_drv/hd.c,294)。
23 void do_rd_request(void)
24 {
25         int     len;
26         char    *addr;
27
28         INIT_REQUEST;       // 检测请求的合法性(参见 kernel/blk_drv/blk.h,127)。
   // 下面语句取得 ramdisk 的起始扇区对应的内存起始位置和内存长度。
   // 其中 sector << 9 表示 sector * 512，CURRENT 定义为(blk_dev[MAJOR_NR].current_request)。
29         addr = rd_start + (CURRENT->sector << 9);
30         len = CURRENT->nr_sectors << 9;
   // 如果子设备号不为 1 或者对应内存起始位置>虚拟盘末尾，则结束该请求，并跳转到 repeat 处
   // （定义在 28 行的 INIT_REQUEST 内开始处）。
31         if ((MINOR(CURRENT->dev) != 1) || (addr+len > rd_start+rd_length)) {
32                 end_request(0);
33                 goto repeat;
34         }
   // 如果是写命令(WRITE)，则将请求项中缓冲区的内容复制到 addr 处，长度为 len 字节。
35         if (CURRENT-> cmd == WRITE) {
36                 (void ) memcpy(addr,
37                                CURRENT->buffer,
38                                len);
   // 如果是读命令(READ)，则将 addr 开始的内容复制到请求项中缓冲区中，长度为 len 字节。
39         } else if (CURRENT->cmd == READ) {
40                 (void) memcpy(CURRENT->buffer,
41                               addr,
42                               len);
   // 否则显示命令不存在，死机。
43         } else
44                 panic("unknown ramdisk-command");
   // 请求项成功后处理，置更新标志。并继续处理本设备的下一请求项。
45         end_request(1);
46         goto repeat;
47 }
48
49 /*
50  * Returns amount of memory which needs to be reserved.
51  */
   /* 返回内存虚拟盘 ramdisk 所需的内存量 */
   // 虚拟盘初始化函数。确定虚拟盘在内存中的起始地址，长度。并对整个虚拟盘区清零。
52 long rd_init(long mem_start, int length)
53 {
54         int     i;
55         char    *cp;
56
57         blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;  // do_rd_request()。
58         rd_start = (char *) mem_start;
```

```
        rd_length = length;
        cp = rd_start;
        for (i=0; i < length; i++)
                *cp++ = '\0';
        return(length);
}

/*
 * If the root device is the ram disk, try to load it.
 * In order to do this, the root device is originally set to the
 * floppy, and we later change it to be ram disk.
 */
/*
 * 如果根文件系统设备(root device)是 ramdisk 的话，则尝试加载它。root device 原先是指向
 * 软盘的，我们将它改成指向 ramdisk。
 */
//// 加载根文件系统到 ramdisk。
void rd_load(void)
{
        struct buffer_head *bh;
        struct super_block      s;
        int             block = 256;    /* Start at block 256 */
        int             i = 1;
        int             nblocks;
        char            *cp;            /* Move pointer */

        if (!rd_length)                 // 如果 ramdisk 的长度为零，则退出。
                return;
        printk("Ram disk: %d bytes, starting at 0x%x\n", rd_length,
                (int) rd_start);        // 显示 ramdisk 的大小以及内存起始位置。
        if (MAJOR(ROOT_DEV) != 2)       // 如果此时根文件设备不是软盘，则退出。
                return;
// 读软盘块 256+1,256,256+2。breada()用于读取指定的数据块，并标出还需要读的块，然后返回
// 含有数据块的缓冲区指针。如果返回 NULL，则表示数据块不可读(fs/buffer.c,322)。
// 这里 block+1 是指磁盘上的超级块。
        bh = breada(ROOT_DEV,block+1,block,block+2,-1);
        if (!bh) {
                printk("Disk error while looking for ramdisk!\n");
                return;
        }
// 将 s 指向缓冲区中的磁盘超级块。(d_super_block 磁盘中超级块结构)。
        *((struct d_super_block *) &s) = *((struct d_super_block *) bh->b_data);
        brelse(bh);                             // [?? 为什么数据没有复制就立刻释放呢？]
        if (s.s_magic != SUPER_MAGIC)   // 如果超级块中魔数不对，则说明不是 minix 文件系统。
                /* No ram disk image present, assume normal floppy boot */
                /* 磁盘中没有 ramdisk 映像文件，退出执行通常的软盘引导 */
                return;
// 块数 = 逻辑块数(区段数) * 2^(每区段块数的次方)。
// 如果数据块数大于内存中虚拟盘所能容纳的块数，则不能加载，显示出错信息并返回。否则显示
// 加载数据块信息。
        nblocks = s.s_nzones << s.s_log_zone_size;
        if (nblocks > (rd_length >> BLOCK_SIZE_BITS)) {
                printk("Ram disk image too big!  (%d blocks, %d avail)\n",
```

```
                            nblocks, rd_length >> BLOCK_SIZE_BITS);
                    return;
            }
            printk("Loading %d bytes into ram disk... 0000k",
                    nblocks << BLOCK_SIZE_BITS);
    // cp 指向虚拟盘起始处，然后将磁盘上的根文件系统映象文件复制到虚拟盘上。
            cp = rd_start;
            while (nblocks) {
                    if (nblocks > 2)   // 如果需读取的块数多于 3 快则采用超前预读方式读数据块。
                            bh = breada(ROOT_DEV, block, block+1, block+2, -1);
                    else               // 否则就单块读取。
                            bh = bread(ROOT_DEV, block);
                    if (!bh) {
                            printk("I/O error on block %d, aborting load\n",
                                    block);
                            return;
                    }
                    (void) memcpy(cp, bh->b_data, BLOCK_SIZE); // 将缓冲区中的数据复制到 cp 处。
                    brelse(bh);                                // 释放缓冲区。
                    printk("\010\010\010\010\010%4dk",i);      // 打印加载块计数值。
                    cp += BLOCK_SIZE;                          // 虚拟盘指针前移。
                    block++;
                    nblocks--;
                    i++;
            }
            printk("\010\010\010\010\010done \n");
            ROOT_DEV=0x0101;                        // 修改 ROOT_DEV 使其指向虚拟盘 ramdisk。
    }

```

6.7 floppy.c 文件

6.7.1 功能描述

6.7.2 代码注释

列表 6.7 linux/kernel/blk_drv/floppy.c 程序

```
/*
 *  linux/kernel/floppy.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 02.12.91 - Changed to static variables to indicate need for reset
 * and recalibrate. This makes some things easier (output_byte reset
 * checking etc), and means less interrupt jumping in case of errors,
 * so the code is hopefully easier to understand.
```

```
 */
/*
 * 02.12.91 - 修改成静态变量，以适应复位和重新校正操作。这使得某些事情
 * 做起来较为方便（output_byte 复位检查等），并且意味着在出错时中断跳转
 * 要少一些，所以希望代码能更容易被理解。
 */

/*
 * This file is certainly a mess. I've tried my best to get it working,
 * but I don't like programming floppies, and I have only one anyway.
 * Urgel. I should check for more errors, and do more graceful error
 * recovery. Seems there are problems with several drives. I've tried to
 * correct them. No promises.
 */
/*
 * 这个文件当然比较混乱。我已经尽我所能使其能够工作，但我不喜欢软驱编程，
 * 而且我也只有一个软驱。另外，我应该做更多的查错工作，以及改正更多的错误。
 * 对于某些软盘驱动器好象还存在一些问题。我已经尝试着进行纠正了，但不能保证
 * 问题已消失。
 */

/*
 * As with hd.c, all routines within this file can (and will) be called
 * by interrupts, so extreme caution is needed. A hardware interrupt
 * handler may not sleep, or a kernel panic will happen. Thus I cannot
 * call "floppy-on" directly, but have to set a special timer interrupt
 * etc.
 *
 * Also, I'm not certain this works on more than 1 floppy. Bugs may
 * abund.
 */
/*
 * 如同 hd.c 文件一样，该文件中的所有子程序都能够被中断调用，所以需要特别
 * 地小心。硬件中断处理程序是不能睡眠的，否则内核就会傻掉(死机)☺。因此不能
 * 直接调用"floppy-on"，而只能设置一个特殊的时间中断等。
 *
 * 另外，我不能保证该程序能在多于 1 个软驱的系统上工作，有可能存在错误。
 */

#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/fs.h>     // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <linux/fdreg.h>  // 软驱头文件。含有软盘控制器参数的一些定义。
#include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
#include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
#include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

#define MAJOR_NR 2        // 软驱的主设备号是 2。
#include "blk.h"          // 块设备头文件。定义请求数据结构、块设备数据结构和宏函数等信息。

static int recalibrate = 0;         // 标志：需要重新校正。
static int reset = 0;               // 标志：需要进行复位操作。
```

```
static int seek = 0;                // 寻道。

extern unsigned char current_DOR;   // 当前数字输出寄存器(Digital Output Register)。

#define immoutb_p(val,port) \       // 字节直接输出（嵌入汇编语言宏）。
__asm__("outb %0,%1\n\tjmp 1f\n1:\tjmp 1f\n1:"::"a" ((char) (val)),"i" (port))

// 这两个定义用于计算软驱的设备号。次设备号 = TYPE*4 + DRIVE。计算方法参见列表后。
#define TYPE(x) ((x)>>2)            // 软驱类型（2--1.2Mb，7--1.44Mb）。
#define DRIVE(x) ((x)&0x03)         // 软驱序号（0--3 对应 A--D）。
/*
 * Note that MAX_ERRORS=8 doesn't imply that we retry every bad read
 * max 8 times - some types of errors increase the errorcount by 2,
 * so we might actually retry only 5-6 times before giving up.
 */
/*
 * 注意，下面定义 MAX_ERRORS=8 并不表示对每次读错误尝试最多 8 次 - 有些类型
 * 的错误将把出错计数值乘 2，所以我们实际上在放弃操作之前只需尝试 5-6 遍即可。
 */
#define MAX_ERRORS 8

/*
 * globals used by 'result()'
 */
/* 下面是函数'result()'使用的全局变量 */
// 这些状态字节中各比特位的含义请参见 include/linux/fdreg.h 头文件。
#define MAX_REPLIES 7                              // FDC 最多返回 7 字节的结果信息。
static unsigned char reply_buffer[MAX_REPLIES];    // 存放 FDC 返回的结果信息。
#define ST0 (reply_buffer[0])                      // 返回结果状态字节 0。
#define ST1 (reply_buffer[1])                      // 返回结果状态字节 1。
#define ST2 (reply_buffer[2])                      // 返回结果状态字节 2。
#define ST3 (reply_buffer[3])                      // 返回结果状态字节 3。

/*
 * This struct defines the different floppy types. Unlike minix
 * linux doesn't have a "search for right type"-type, as the code
 * for that is convoluted and weird. I've got enough problems with
 * this driver as it is.
 *
 * The 'stretch' tells if the tracks need to be boubled for some
 * types (ie 360kB diskette in 1.2MB drive etc). Others should
 * be self-explanatory.
 */
/*
 * 下面的软盘结构定义了不同的软盘类型。与 minix 不同的是，linux 没有
 * "搜索正确的类型"-类型，因为对其处理的代码令人费解且怪怪的。本程序
 * 已经让我遇到了许多的问题了。
 *
 * 对某些类型的软盘（例如在 1.2MB 驱动器中的 360kB 软盘等），'stretch'用于
 * 检测磁道是否需要特殊处理。其它参数应该是自明的。
 */
// 软盘参数有：
// size          大小(扇区数)；
```

```
// sect          每磁道扇区数;
// head          磁头数;
// track         磁道数;
// stretch       对磁道是否要特殊处理（标志）;
// gap           扇区间隙长度(字节数);
// rate          数据传输速率;
// spec1         参数（高 4 位步进速率，低四位磁头卸载时间）。
static struct floppy_struct {
        unsigned int size, sect, head, track, stretch;
        unsigned char gap,rate,spec1;
} floppy_type[] = {
        {    0, 0,0, 0,0,0x00,0x00,0x00 },      /* no testing */
        {  720, 9,2,40,0,0x2A,0x02,0xDF },      /* 360kB PC diskettes */
        { 2400,15,2,80,0,0x1B,0x00,0xDF },      /* 1.2 MB AT-diskettes */
        {  720, 9,2,40,1,0x2A,0x02,0xDF },      /* 360kB in 720kB drive */
        { 1440, 9,2,80,0,0x2A,0x02,0xDF },      /* 3.5" 720kB diskette */
        {  720, 9,2,40,1,0x23,0x01,0xDF },      /* 360kB in 1.2MB drive */
        { 1440, 9,2,80,0,0x23,0x01,0xDF },      /* 720kB in 1.2MB drive */
        { 2880,18,2,80,0,0x1B,0x00,0xCF },      /* 1.44MB diskette */
};
/*
 * Rate is 0 for 500kb/s, 2 for 300kbps, 1 for 250kbps
 * Spec1 is 0xSH, where S is stepping rate (F=1ms, E=2ms, D=3ms etc),
 * H is head unload time (1=16ms, 2=32ms, etc)
 *
 * Spec2 is (HLD<<1 | ND), where HLD is head load time (1=2ms, 2=4 ms etc)
 * and ND is set means no DMA. Hardcoded to 6 (HLD=6ms, use DMA).
 */
/*
 * 上面速率 rate：0 表示 500kb/s，1 表示 300kbps，2 表示 250kbps。
 * 参数 spec1 是 0xSH，其中 S 是步进速率（F-1 毫秒，E-2ms，D=3ms 等），
 * H 是磁头卸载时间（1=16ms，2=32ms 等）
 *
 * spec2 是（HLD<<1 | ND），其中 HLD 是磁头加载时间（1=2ms，2=4ms 等）
 * ND 置位表示不使用 DMA（No DMA），在程序中硬编码成 6（HLD=6ms，使用 DMA）。
 */

extern void floppy_interrupt(void);
extern char tmp_floppy_area[1024];

/*
 * These are global variables, as that's the easiest way to give
 * information to interrupts. They are the data used for the current
 * request.
 */
/*
 * 下面是一些全局变量，因为这是将信息传给中断程序最简单的方式。它们是
 * 用于当前请求的数据。
 */
static int cur_spec1 = -1;
static int cur_rate = -1;
static struct floppy_struct * floppy = floppy_type;
static unsigned char current_drive = 0;
```

```
static unsigned char sector = 0;
static unsigned char head = 0;
static unsigned char track = 0;
static unsigned char seek_track = 0;
static unsigned char current_track = 255;
static unsigned char command = 0;
unsigned char selected = 0;
struct task_struct * wait_on_floppy_select = NULL;

//// 释放（取消选定的）软盘（软驱）。
// 数字输出寄存器(DOR)的低 2 位用于指定选择的软驱（0-3 对应 A-D）。
void floppy_deselect(unsigned int nr)
{
        if (nr != (current_DOR & 3))
                printk("floppy_deselect: drive not selected\n\r");
        selected = 0;
        wake_up(&wait_on_floppy_select);
}

/*
 * floppy-change is never called from an interrupt, so we can relax a bit
 * here, sleep etc. Note that floppy-on tries to set current_DOR to point
 * to the desired drive, but it will probably not survive the sleep if
 * several floppies are used at the same time: thus the loop.
 */
/*
 * floppy-change()不是从中断程序中调用的，所以这里我们可以轻松一下，睡觉等。
 * 注意 floppy-on()会尝试设置 current_DOR 指向所需的驱动器，但当同时使用几个
 * 软盘时不能睡眠：因此此时只能使用循环方式。
 */
//// 检测指定软驱中软盘更换情况。如果软盘更换了则返回 1，否则返回 0。
int floppy_change(unsigned int nr)
{
repeat:
        floppy_on(nr);               // 开启指定软驱 nr（kernel/sched.c,251）。
// 如果当前选择的软驱不是指定的软驱 nr，并且已经选择其它了软驱，则让当前任务进入可中断
// 等待状态。
        while ((current_DOR & 3) != nr && selected)
                interruptible_sleep_on(&wait_on_floppy_select);
// 如果当前没有选择其它软驱或者当前任务被唤醒时，当前软驱仍然不是指定的软驱 nr，则循环等待。
        if ((current_DOR & 3) != nr)
                goto repeat;
// 取数字输入寄存器值，如果最高位（位 7）置位，则表示软盘已更换，此时关闭马达并退出返回 1。
// 否则关闭马达退出返回 0。
        if (inb(FD_DIR) & 0x80) {
                floppy_off(nr);
                return 1;
        }
        floppy_off(nr);
        return 0;
}

//// 复制内存块。
```

```
#define copy_buffer(from,to) \
__asm__("cld ; rep ; movsl" \
        ::"c" (BLOCK_SIZE/4),"S" ((long)(from)),"D" ((long)(to)) \
        :"cx","di","si")

//// 设置（初始化）软盘 DMA 通道。
static void setup_DMA(void)
{
        long addr = (long) CURRENT->buffer;  // 当前请求项缓冲区所处内存中位置（地址）。

        cli();
// 如果缓冲区处于内存 1M 以上的地方，则将 DMA 缓冲区设在临时缓冲区域(tmp_floppy_area 数组)
// (因为 8237A 芯片只能在 1M 地址范围内寻址)。如果是写盘命令，则还需将数据复制到该临时区域。
        if (addr >= 0x100000) {
                addr = (long) tmp_floppy_area;
                if (command == FD_WRITE)
                        copy_buffer(CURRENT->buffer,tmp_floppy_area);
        }
/* mask DMA 2 */    /* 屏蔽 DMA 通道 2 */
// 单通道屏蔽寄存器端口为 0x10。位 0-1 指定 DMA 通道(0--3)，位 2：1 表示屏蔽，0 表示允许请求。
        immoutb_p(4|2,10);
/* output command byte. I don't know why, but everyone (minix, */
/* sanches & canton) output this twice, first to 12 then to 11 */
/* 输出命令字节。我是不知道为什么，但是每个人（minix，*/
/* sanches 和 canton）都输出两次，首先是 12 口，然后是 11 口 */
// 下面嵌入汇编代码向 DMA 控制器端口 12 和 11 写方式字（读盘 0x46，写盘 0x4A）。
        __asm__("outb %%al,$12\n\tjmp 1f\n1:\tjmp 1f\n1:\t"
        "outb %%al,$11\n\tjmp 1f\n1:\tjmp 1f\n1:"::
        "a" ((char) ((command == FD_READ)?DMA_READ:DMA_WRITE)));
/* 8 low bits of addr */    /* 地址低 0-7 位 */
// 向 DMA 通道 2 写入基/当前地址寄存器（端口 4）。
        immoutb_p(addr,4);
        addr >>= 8;
/* bits 8-15 of addr */    /* 地址高 8-15 位 */
        immoutb_p(addr,4);
        addr >>= 8;
/* bits 16-19 of addr */    /* 地址 16-19 位 */
// DMA 只可以在 1M 内存空间内寻址，其高 16-19 位地址需放入页面寄存器(端口 0x81)。
        immoutb_p(addr,0x81);
/* low 8 bits of count-1 (1024-1=0x3ff) */    /* 计数器低 8 位(1024-1=0x3ff) */
// 向 DMA 通道 2 写入基/当前字节计数器值（端口 5）。
        immoutb_p(0xff,5);
/* high 8 bits of count-1 */    /* 计数器高 8 位 */
// 一次共传输 1024 字节（两个扇区）。
        immoutb_p(3,5);
/* activate DMA 2 */    /* 开启 DMA 通道 2 的请求 */
// 复位对 DMA 通道 2 的屏蔽，开放 DMA2 请求 DREQ 信号。
        immoutb_p(0|2,10);
        sti();
}

//// 向软盘控制器输出一个字节数据（命令或参数）。
static void output_byte(char byte)
```

```
{
        int counter;
        unsigned char status;

        if (reset)
                return;
// 循环读取主状态控制器 FD_STATUS(0x3f4)的状态。如果状态是 STATUS_READY 并且 STATUS_DIR=0
// (CPU→FDC)，则向数据端口输出指定字节。
        for(counter = 0 ; counter < 10000 ; counter++) {
                status = inb_p(FD_STATUS) & (STATUS_READY | STATUS_DIR);
                if (status == STATUS_READY) {
                        outb(byte,FD_DATA);
                        return;
                }
        }
// 如果到循环 1 万次结束还不能发送，则置复位标志，并打印出错信息。
        reset = 1;
        printk("Unable to send byte to FDC\n\r");
}

//// 读取 FDC 执行的结果信息。
// 结果信息最多 7 个字节，存放在 reply_buffer[]中。返回读入的结果字节数，若返回值=-1
// 表示出错。
static int result(void)
{
        int i = 0, counter, status;

        if (reset)
                return -1;
        for (counter = 0 ; counter < 10000 ; counter++) {
                status = inb_p(FD_STATUS)&(STATUS_DIR|STATUS_READY|STATUS_BUSY);
                if (status == STATUS_READY)
                        return i;
                if (status == (STATUS_DIR|STATUS_READY|STATUS_BUSY)) {
                        if (i >= MAX_REPLIES)
                                break;
                        reply_buffer[i++] = inb_p(FD_DATA);
                }
        }
        reset = 1;
        printk("Getstatus times out\n\r");
        return -1;
}

//// 软盘操作出错中断调用函数。由软驱中断处理程序调用。
static void bad_flp_intr(void)
{
        CURRENT->errors++;          // 当前请求项出错次数增 1。
// 如果当前请求项出错次数大于最大允许出错次数，则取消选定当前软驱，并结束该请求项（不更新）。
        if (CURRENT->errors > MAX_ERRORS) {
                floppy_deselect(current_drive);
                end_request(0);
        }
```

```
// 如果当前请求项出错次数大于最大允许出错次数的一半，则置复位标志，需对软驱进行复位操作，
// 然后再试。否则软驱需重新校正一下，再试。
        if (CURRENT->errors > MAX_ERRORS/2)
                reset = 1;
        else
                recalibrate = 1;
}

/*
 * Ok, this interrupt is called after a DMA read/write has succeeded,
 * so we check the results, and copy any buffers.
 */
/*
 * OK，下面该中断处理函数是在 DMA 读/写成功后调用的，这样我们就可以检查执行结果，
 * 并复制缓冲区中的数据。
 */
//// 软盘读写操作成功中断调用函数。。
static void rw_interrupt(void)
{
// 如果返回结果字节数不等于 7，或者状态字节 0、1 或 2 中存在出错标志，则若是写保护
// 就显示出错信息，释放当前驱动器，并结束当前请求项。否则就执行出错计数处理。
// 然后继续执行软盘请求操作。
// ( 0xf8 = ST0_INTR | ST0_SE | ST0_ECE | ST0_NR )
// ( 0xbf = ST1_EOC | ST1_CRC | ST1_OR | ST1_ND | ST1_WP | ST1_MAM，应该是 0xb7)
// ( 0x73 = ST2_CM | ST2_CRC | ST2_WC | ST2_BC | ST2_MAM )
        if (result() != 7 || (ST0 & 0xf8) || (ST1 & 0xbf) || (ST2 & 0x73)) {
                if (ST1 & 0x02) {          // 0x02 = ST1_WP - Write Protected。
                        printk("Drive %d is write protected\n\r",current_drive);
                        floppy_deselect(current_drive);
                        end_request(0);
                } else
                        bad_flp_intr();
                do_fd_request();
                return;
        }
// 如果当前请求项的缓冲区位于 1M 地址以上，则说明此次软盘读操作的内容还放在临时缓冲区内，
// 需要复制到请求项的缓冲区中（因为 DMA 只能在 1M 地址范围寻址）。
        if (command == FD_READ && (unsigned long)(CURRENT->buffer) >= 0x100000)
                copy_buffer(tmp_floppy_area,CURRENT->buffer);
// 释放当前软盘，结束当前请求项（置更新标志），再继续执行其它软盘请求项。
        floppy_deselect(current_drive);
        end_request(1);
        do_fd_request();
}

//// 设置 DMA 并输出软盘操作命令和参数（输出 1 字节命令+ 0~7 字节参数）。
inline void setup_rw_floppy(void)
{
        setup_DMA();                    // 初始化软盘 DMA 通道。
        do_floppy = rw_interrupt;       // 置软盘中断调用函数指针。
        output_byte(command);           // 发送命令字节。
        output_byte(head<<2 | current_drive);  // 发送参数（磁头号+驱动器号）。
        output_byte(track);             // 发送参数（磁道号）。
```

```
        output_byte(head);          // 发送参数（磁头号）。
        output_byte(sector);        // 发送参数（起始扇区号）。
        output_byte(2);         /* sector size = 512 */   // 发送参数(字节数(N=2)512 字节)。
        output_byte(floppy->sect); // 发送参数（每磁道扇区数）。
        output_byte(floppy->gap);  // 发送参数（扇区间隔长度）。
        output_byte(0xFF);      /* sector size (0xff when n!=0 ?) */
                                    // 发送参数（当 N=0 时，扇区定义的字节长度），这里无用。
// 若在发送命令和参数时发生错误，则继续执行下一软盘操作请求。
        if (reset)
                do_fd_request();
}

/*
 * This is the routine called after every seek (or recalibrate) interrupt
 * from the floppy controller. Note that the "unexpected interrupt" routine
 * also does a recalibrate, but doesn't come here.
 */
/*
 * 该子程序是在每次软盘控制器寻道（或重新校正）中断后被调用的。注意
 * "unexpected interrupt"(意外中断)子程序也会执行重新校正操作，但不在此地。
 */
//// 寻道处理中断调用函数。
// 首先发送检测中断状态命令，获得状态信息 ST0 和磁头所在磁道信息。若出错则执行错误计数
// 检测处理或取消本次软盘操作请求项。否则根据状态信息设置当前磁道变量，然后调用函数
// setup_rw_floppy()设置 DMA 并输出软盘读写命令和参数。
static void seek_interrupt(void)
{
/* sense drive status */    /* 检测中断状态 */
// 发送检测中断状态命令，该命令不带参数。返回结果信息两个字节：ST0 和磁头当前磁道号。
        output_byte(FD_SENSEI);
// 如果返回结果字节数不等于 2，或者 ST0 不为寻道结束，或者磁头所在磁道(ST1)不等于设定磁道，
// 则说明发生了错误，于是执行检测错误计数处理，然后继续执行软盘请求项，并退出。
        if (result() != 2 || (ST0 & 0xF8) != 0x20 || ST1 != seek_track) {
                bad_flp_intr();
                do_fd_request();
                return;
        }
        current_track = ST1;    // 设置当前磁道。
        setup_rw_floppy();     // 设置 DMA 并输出软盘操作命令和参数。
}

/*
 * This routine is called when everything should be correctly set up
 * for the transfer (ie floppy motor is on and the correct floppy is
 * selected).
 */
/*
 * 该函数是在传输操作的所有信息都正确设置好后被调用的（也即软驱马达已开启
 * 并且已选择了正确的软盘（软驱）。
 */
//// 读写数据传输函数。

static void transfer(void)
```

```
{
// 首先看当前驱动器参数是否就是指定驱动器的参数，若不是就发送设置驱动器参数命令及相应
// 参数（参数 1：高 4 位步进速率，低四位磁头卸载时间；参数 2：磁头加载时间）。
        if (cur_spec1 != floppy->spec1) {
                cur_spec1 = floppy->spec1;
                output_byte(FD_SPECIFY);        // 发送设置磁盘参数命令。
                output_byte(cur_spec1);         /* hut etc */   // 发送参数。
                output_byte(6);                 /* Head load time =6ms, DMA */
        }
// 判断当前数据传输速率是否与指定驱动器的一致，若不是就发送指定软驱的速率值到数据传输
// 速率控制寄存器(FD_DCR)。
        if (cur_rate != floppy->rate)
                outb_p(cur_rate = floppy->rate,FD_DCR);
// 若返回结果信息表明出错，则再调用软盘请求函数，并返回。
        if (reset) {
                do_fd_request();
                return;
        }
// 若寻道标志为零（不需要寻道），则设置 DMA 并发送相应读写操作命令和参数，然后返回。
        if (!seek) {
                setup_rw_floppy();
                return;
        }
// 否则执行寻道处理。置软盘中断处理调用函数为寻道中断函数。
        do_floppy = seek_interrupt;
// 如果器始磁道号不等于零则发送磁头寻道命令和参数
        if (seek_track) {
                output_byte(FD_SEEK);     // 发送磁头寻道命令。
                output_byte(head<<2 | current_drive);  //发送参数：磁头号+当前软驱号。
                output_byte(seek_track);               // 发送参数：磁道号。
        } else {
                output_byte(FD_RECALIBRATE);           // 发送重新校正命令。
                output_byte(head<<2 | current_drive);  //发送参数：磁头号+当前软驱号。
        }
// 如果复位标志已置位，则继续执行软盘请求项。
        if (reset)
                do_fd_request();
}

/*
 * Special case - used after a unexpected interrupt (or reset)
 */
/*
 * 特殊情况 - 用于意外中断（或复位）处理后。
 */
//// 软驱重新校正中断调用函数。
// 首先发送检测中断状态命令（无参数），如果返回结果表明出错，则置复位标志，否则复位重新
// 校正标志。然后再次执行软盘请求。
static void recal_interrupt(void)
{
        output_byte(FD_SENSEI);                   // 发送检测中断状态命令。
        if (result()!=2 || (ST0 & 0xE0) == 0x60) // 如果返回结果字节数不等于 2 或命令
                reset = 1;                        // 异常结束，则置复位标志。
```

```
        else                                    // 否则复位重新校正标志。
                recalibrate = 0;
        do_fd_request();                        // 执行软盘请求项。
}

//// 意外软盘中断请求中断调用函数。
// 首先发送检测中断状态命令（无参数），如果返回结果表明出错，则置复位标志，否则置重新
// 校正标志。
void unexpected_floppy_interrupt(void)
{
        output_byte(FD_SENSEI);                 // 发送检测中断状态命令。
        if (result()!=2 || (ST0 & 0xE0) == 0x60) // 如果返回结果字节数不等于 2 或命令
                reset = 1;                      // 异常结束，则置复位标志。
        else                                    // 否则置重新校正标志。
                recalibrate = 1;
}

//// 软盘重新校正处理函数。
// 向软盘控制器 FDC 发送重新校正命令和参数，并复位重新校正标志。
static void recalibrate_floppy(void)
{
        recalibrate = 0;                // 复位重新校正标志。
        current_track = 0;              // 当前磁道号归零。
        do_floppy = recal_interrupt;    // 置软盘中断调用函数指针指向重新校正调用函数。
        output_byte(FD_RECALIBRATE);    // 发送命令：重新校正。
        output_byte(head<<2 | current_drive); // 发送参数：（磁头号加）当前驱动器号。
        if (reset)                      // 如果出错(复位标志被置位)则继续执行软盘请求。
                do_fd_request();
}

//// 软盘控制器 FDC 复位中断调用函数。在软盘中断处理程序中调用。
// 首先发送检测中断状态命令（无参数），然后读出返回的结果字节。接着发送设定软驱参数命令
// 和相关参数，最后再次调用执行软盘请求。
static void reset_interrupt(void)
{
        output_byte(FD_SENSEI);                 // 发送检测中断状态命令。
        (void) result();                        // 读取命令执行结果字节。
        output_byte(FD_SPECIFY);                // 发送设定软驱参数命令。
        output_byte(cur_spec1);         /* hut etc */ // 发送参数。
        output_byte(6);                 /* Head load time =6ms, DMA */
        do_fd_request();                        // 调用执行软盘请求。
}

/*
 * reset is done by pulling bit 2 of DOR low for a while.
 */
/* FDC 复位是通过将数字输出寄存器(DOR)位 2 置 0 一会儿实现的 */
//// 复位软盘控制器。
static void reset_floppy(void)
{
        int i;

        reset = 0;                              // 复位标志置 0。
```

```
        cur_spec1 = -1;
        cur_rate = -1;
        recalibrate = 1;                    // 重新校正标志置位。
        printk("Reset-floppy called\n\r");  // 显示执行软盘复位操作信息。
        cli();                              // 关中断。
        do_floppy = reset_interrupt;        // 设置在软盘中断处理程序中调用的函数。
        outb_p(current_DOR & ~0x04,FD_DOR); // 对软盘控制器 FDC 执行复位操作。
        for (i=0 ; i<100 ; i++)             // 空操作，延迟。
                __asm__("nop");
        outb(current_DOR,FD_DOR);           // 再启动软盘控制器。
        sti();                              // 开中断。
}

//// 软驱启动定时中断调用函数。
// 首先检查数字输出寄存器(DOR)，使其选择当前指定的驱动器。然后调用执行软盘读写传输
// 函数 transfer()。
static void floppy_on_interrupt(void)
{
/* We cannot do a floppy-select, as that might sleep. We just force it */
/* 我们不能任意设置选择的软驱，因为这样做可能会引起进程睡眠。我们只是迫使它自己选择 */
        selected = 1;   // 置已选择当前驱动器标志。
// 如果当前驱动器号与数字输出寄存器 DOR 中的不同，则重新设置 DOR 为当前驱动器 current_drive。
// 定时延迟 2 个滴答时间，然后调用软盘读写传输函数 transfer()。否则直接调用软盘读写传输函数。
        if (current_drive != (current_DOR & 3)) {
                current_DOR &= 0xFC;
                current_DOR |= current_drive;
                outb(current_DOR,FD_DOR);           // 向数字输出寄存器输出当前 DOR。
                add_timer(2,&transfer);             // 添加定时器并执行传输函数。
        } else
                transfer();                        // 执行软盘读写传输函数。
}

//// 软盘读写请求项处理函数。
//
void do_fd_request(void)
{
        unsigned int block;

        seek = 0;
// 如果复位标志已置位，则执行软盘复位操作，并返回。
        if (reset) {
                reset_floppy();
                return;
        }
// 如果重新校正标志已置位，则执行软盘重新校正操作，并返回。
        if (recalibrate) {
                recalibrate_floppy();
                return;
        }
// 检测请求项的合法性(参见 kernel/blk_drv/blk.h,127)。
        INIT_REQUEST;
// 将请求项结构中软盘设备号中的软盘类型(MINOR(CURRENT->dev)>>2)作为索引取得软盘参数块。
        floppy = (MINOR(CURRENT->dev)>>2) + floppy_type;
```

```
    // 如果当前驱动器不是请求项中指定的驱动器，则置标志 seek，表示需要进行寻道操作。
    // 然后置请求项设备为当前驱动器。
432         if (current_drive != CURRENT_DEV)
433                 seek = 1;
434         current_drive = CURRENT_DEV;
    // 设置读写起始扇区。因为每次读写是以块为单位（1 块 2 个扇区），所以起始扇区需要起码比
    // 磁盘总扇区数小 2 个扇区。否则结束该次软盘请求项，执行下一个请求项。
435         block = CURRENT->sector;          // 取当前软盘请求项中起始扇区号→block。
436         if (block+2 > floppy->size) {     // 如果 block+2 大于磁盘扇区总数，则
437                 end_request(0);           // 结束本次软盘请求项。
438                 goto repeat;
439         }
    // 求对应在磁道上的扇区号，磁头号，磁道号，搜寻磁道号（对于软驱读不同格式的盘）。
440         sector = block % floppy->sect;    // 起始扇区对每磁道扇区数取模，得磁道上扇区号。
441         block /= floppy->sect;            // 起始扇区对每磁道扇区数取整，得起始磁道数。
442         head = block % floppy->head;      // 起始磁道数对磁头数取模，得操作的磁头号。
443         track = block / floppy->head;     // 起始磁道数对磁头数取整，得操作的磁道号。
444         seek_track = track << floppy->stretch; // 相应于驱动器中盘类型进行调整，得寻道号。
    // 如果寻道号与当前磁头所在磁道不同，则置需要寻道标志 seek。
445         if (seek_track != current_track)
446                 seek = 1;
447         sector++;                         // 磁盘上实际扇区计数是从 1 算起。
448         if (CURRENT->cmd == READ)         // 如果请求项中是读操作，则置软盘读命令码。
449                 command = FD_READ;
450         else if (CURRENT->cmd == WRITE)   // 如果请求项中是写操作，则置软盘写命令码。
451                 command = FD_WRITE;
452         else
453                 panic("do_fd_request: unknown command");
    // 添加定时器，用于指定驱动器到能正常运行所需延迟的时间（滴答数），当定时时间到时就调用
    // 函数 floppy_on_interrupt()，
454         add_timer(ticks_to_floppy_on(current_drive),&floppy_on_interrupt);
455 }
456
    //// 软盘系统初始化。
    // 设置软盘块设备的请求处理函数(do_fd_request())，并设置软盘中断门(int 0x26，对应硬件
    // 中断请求信号 IRQ6），然后取消对该中断信号的屏蔽，允许软盘控制器 FDC 发送中断请求信号。
457 void floppy_init(void)
458 {
459         blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;  // = do_fd_request()。
460         set_trap_gate(0x26,&floppy_interrupt);   //设置软盘中断门 int 0x26(38)。
461         outb(inb_p(0x21)&~0x40,0x21);            // 复位软盘的中断请求屏蔽位，允许
                                                     // 软盘控制器发送中断请求信号。
462 }
463
```

6.7.3 其它信息

6.7.3.1 软盘驱动器的设备号

在 Linux 中，软驱的主设备号是 2，次设备号 = TYPE*4 + DRIVE，其中 DRIVE 为 0-3，分别对应软驱 A、B、C 或 D；TYPE 是软驱的类型，2 表示 1.2M 软驱，7 表示 1.44M 软驱，也即 floppy.c 中 85 行定义的软盘类型（floppy_type[]）数组的索引值：

0　　不用；
1　　360kB PC 软驱；

2　　1.2MB AT 软驱；
3　　360kB 在 720kB 驱动器中使用；
4　　3.5″ 720kB 软盘；
5　　360kB 在 1.2MB 驱动器中使用；
6　　720kB 在 1.2MB 驱动器中使用；
7　　1.44MB 软驱。

例如，因为 7*4 + 0 = 28，所以 /dev/PS0 (2,28)指的是 1.44M A 驱动器,其设备号是 0x021c。

同理 /dev/at0 (2,8)指的是 1.2M A 驱动器，其设备号是 0x0208。

6.7.3.2 软盘控制器编程方法

对软盘控制器的编程比较烦琐。在编程时需要访问 4 个端口，分别对应一个或多个寄存器。对于 1.2M 的软盘控制器有以下一些端口。

表 6.9 软盘控制器端口

I/O 端口	读写性	寄存器名称
0x3f2	只写	数字输出寄存器（DOR）(数字控制寄存器)
0x3f4	只读	FDC 主状态寄存器(STATUS)
0x3f5	读/写	FDC 数据寄存器(DATA)
0x3f7	只读	数字输入寄存器（DIR）
0x3f7	只写	磁盘控制寄存器(DCR)(传输率控制)

数字输出端口 DOR（数字控制端口）是一个 8 位寄存器，它控制驱动器马达开启、驱动器选择、启动/复位 FDC 以及允许/禁止 DMA 及中断请求。

表 6.10 数字输出寄存器定义

D7	D6	D5	D4	D3	D2	D1	D0
启动马达 D	启动马达 C	启动马达 B	启动马达 A	允许请求	启动 FDC	软驱选择	

D7D6D5D4 - 分别控制驱动器 D-A 的马达，1 启动马达；0 关闭马达。
D3　　- 1 允许 DMA 和中断请求；0 禁止 DMA 和中断请求；
D2　　- 1 启动软驱；0 复位软驱；
D1D0　- 00-11 用于选择软盘驱动器 A-D；

FDC 的主状态寄存器也是一个 8 位寄存器，用于反映软盘控制器 FDC 和软盘驱动器 FDD 的基本状态。通常，在 CPU 向 FDC 发送命令之前或从 FDC 获取操作结果之前，都要读取主状态寄存器的状态位，以判别当前 FDC 数据寄存器是否就绪，以及确定数据传送的方向。

表 6.11 FDC 主状态控制器定义

D7	D6	D5	D4	D3	D2	D1	D0
数据口就绪	传送方向	非 DMA 方式	FDC 忙	软驱 D 忙	软驱 C 忙	软驱 B 忙	软驱 A 忙
RQM	DIO	NDM	CB	DDB	DCB	DBB	DAB

D7　　- 1 表示 FDC 数据寄存器已准备就绪；
D6　　- 1 表示数据从 FDC 到 CPU；0 表示数据从 CPU 到 FDC；

D5 - 1 表示 FDC 工作在非 DMA 方式；
D4 - 1 表示 FDC 正处于命令执行忙碌状态；
D3D2D1D0 - 分别代表驱动器 D--A 忙碌状态。

FDC 的数据端口对应多个寄存器（只写型命令寄存器和参数寄存器、只读型结果寄存器），但任一时刻只能有一个寄存器出现在数据端口 0x3f5。在访问只写型寄存器时，主状态控制的 DIO 方向位必须为 0（CPU → FDC），访问只读型寄存器时则反之。在读取结果时只有在 FDC 不忙之后才算读完结果，通常结果数据最多有 7 个字节。

数据输入寄存器（DIR）只有位 7（D7）对软盘有效，用来表示盘片更换状态。其余七位用于硬盘控制器接口。

磁盘控制寄存器(DCR)用于选择盘片在不同类型驱动器上使用的数据传输率。仅使用低 2 位（D1D0），00 - 500kbps，01 - 300kbps，10 - 250kbps。

软盘控制器共可以接受 15 条命令。每个命令均经历三个阶段：命令阶段、执行阶段和结果阶段。

命令阶段是 CPU 向 FDC 发送命令字节和参数字节。每条命令的第一个字节总是命令字节（命令码）。其后跟着 0--8 字节的参数。执行阶段是 FDC 执行命令规定的操作。在执行阶段 CPU 是不加干预的，一般是通过 FDC 发出中断请求获知命令执行的结束。如果 CPU 发出的 FDC 命令是传送数据，则 FDC 可以以中断方式或 DMA 方式进行。中断方式每次传送 1 字节。DMA 方式是在 DMA 控制器管理下，FDC 与内存进行数据的传输直至全部数据传送完。此时 DMA 控制器会将传输字节计数终止信号通知 FDC，最后由 FDC 发出中断请求信号告知 CPU 执行阶段结束。结果阶段是由 CPU 读取 FDC 数据寄存器返回值，从而获得 FDC 命令执行的结果。返回结果数据的长度为 0--7 字节。对于没有返回结果数据的命令，则应向 FDC 发送检测中断状态命令获得操作的状态。

6.7.3.3 DMA 控制器编程

第 7 章 字符设备驱动程序(char driver)

7.1 概述

列表 7.1 linux/kernel/chr_drv 目录

Name	Size	Last modified (GMT)	Description
Makefile	2443 bytes	1991-12-02 03:21:41	m
console.c	14568 bytes	1991-11-23 18:41:21	m
keyboard.S	12780 bytes	1991-12-04 15:07:58	m
rs_io.s	2718 bytes	1991-10-02 14:16:30	m
serial.c	1406 bytes	1991-11-17 21:49:05	m
tty_io.c	7634 bytes	1991-12-08 18:09:15	m
tty_ioctl.c	4979 bytes	1991-11-25 19:59:38	m

7.2 Makefile 文件

7.2.1 功能描述

7.2.2 代码注释

列表 7.2 linux/kernel/chr_drv/Makefile 文件

```
#
# Makefile for the FREAX-kernel character device drivers.
#
# Note! Dependencies are done automagically by 'make dep', which also
# removes any old dependencies. DON'T put your own dependencies here
# unless it's something special (ie not a .c file).
#
# FREAX(Linux)内核字符设备驱动程序的 Makefile 文件。
# 注意！依赖关系是由'make dep'自动进行的，它也会自动去除原来的依赖信息。不要把你自己的
# 依赖关系信息放在这里，除非是特别文件的（也即不是一个.c 文件的信息）。

AR      =gar     # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。
AS      =gas     # GNU 的汇编程序。
LD      =gld     # GNU 的连接程序。
LDFLAGS =-s -x   # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
```

```
13 CC      =gcc    # GNU C 语言编译器。
    # 下一行是 C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
    # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
    # 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
    # 单短小的函数代码嵌入调用程序中；-mstring-insns Linus 自己填加的优化选项，以后不再使用；
    # -nostdinc -I../include 不使用默认路径中的包含文件，而使用指定目录中的(../../include)。
14 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
15         -finline-functions -mstring-insns -nostdinc -I../../include
    # C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
    # 出设备或指定的输出文件中；-nostdinc -I../../include 同前。
16 CPP     =gcc -E -nostdinc -I../../include
17
    # 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
    # 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
    # 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
    # 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。其中$*.s（或$@）是自动目标变量，
    # $<代表第一个先决条件，这里即是符合条件*.c 的文件。
18 .c.s:
19         $(CC) $(CFLAGS) \
20         -S -o $*.s $<
  # 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。22 行是实现该操作的具体命令。
21 .s.o:
22         $(AS) -c -o $*.o $<
23 .c.o:                      # 类似上面，*.c 文件-→*.o 目标文件。不进行连接。
24         $(CC) $(CFLAGS) \
25         -c -o $*.o $<
26
27 OBJS  = tty_io.o console.o keyboard.o serial.o rs_io.o \    # 定义目标文件变量 OBJS。
28         tty_ioctl.o
29
30 chr_drv.a: $(OBJS)         # 在有了先决条件 OBJS 后使用下面的命令连接成目标 chr_drv.a 库文件。
31         $(AR) rcs chr_drv.a $(OBJS)
32         sync
33
   # 对 kerboard.S 汇编程序进行预处理。-traditional 选项用来对程序作修改使其支持传统的 C 编译器。
   # 处理后的程序改名为 kernboard.s。
34 keyboard.s: keyboard.S ../../include/linux/config.h
35         $(CPP) -traditional keyboard.S -o keyboard.s
36
    # 下面的规则用于清理工作。当执行'make clean'时，就会执行下面的命令，去除所有编译
    # 连接生成的文件。'rm'是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
37 clean:
38         rm -f core *.o *.a tmp_make keyboard.s
39         for i in *.c;do rm -f `basename $$i .c`.s;done
40
    # 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
    # 使用字符串编辑程序 sed 对 Makefile 文件（即是本文件）进行处理，输出为删除 Makefile
    # 文件中'### Dependencies'行后面的所有行（下面从 48 开始的行），并生成 tmp_make
    # 临时文件（44 行的作用）。然后对 kernel/chr_drv/目录下的每个 C 文件执行 gcc 预处理操作.
    # -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
    # 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
    # 文件名加上其依赖关系--该源文件中包含的所有头文件列表。把预处理结果都添加到临时
    # 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。
```

```
dep:
        sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
        (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,'`" "; \
                $(CPP) -M $$i;done) >> tmp_make
        cp tmp_make Makefile

### Dependencies:
console.s console.o : console.c ../../include/linux/sched.h \
  ../../include/linux/head.h ../../include/linux/fs.h \
  ../../include/sys/types.h ../../include/linux/mm.h ../../include/signal.h \
  ../../include/linux/tty.h ../../include/termios.h ../../include/asm/io.h \
  ../../include/asm/system.h
serial.s serial.o : serial.c ../../include/linux/tty.h ../../include/termios.h \
  ../../include/linux/sched.h ../../include/linux/head.h \
  ../../include/linux/fs.h ../../include/sys/types.h ../../include/linux/mm.h \
  ../../include/signal.h ../../include/asm/system.h ../../include/asm/io.h
tty_io.s tty_io.o : tty_io.c ../../include/ctype.h ../../include/errno.h \
  ../../include/signal.h ../../include/sys/types.h \
  ../../include/linux/sched.h ../../include/linux/head.h \
  ../../include/linux/fs.h ../../include/linux/mm.h ../../include/linux/tty.h \
  ../../include/termios.h ../../include/asm/segment.h \
  ../../include/asm/system.h
tty_ioctl.s tty_ioctl.o : tty_ioctl.c ../../include/errno.h ../../include/termios.h \
  ../../include/linux/sched.h ../../include/linux/head.h \
  ../../include/linux/fs.h ../../include/sys/types.h ../../include/linux/mm.h \
  ../../include/signal.h ../../include/linux/kernel.h \
  ../../include/linux/tty.h ../../include/asm/io.h \
  ../../include/asm/segment.h ../../include/asm/system.h
```

7.3 serial.c 文件

7.3.1 功能描述

7.3.2 代码注释

列表 7.3 linux/kernel/chr_drv/serial.c 程序

```
/*
 *  linux/kernel/serial.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      serial.c
 *
 * This module implements the rs232 io functions
 *      void rs_write(struct tty_struct * queue);
 *      void rs_init(void);
```

```
 * and all interrupts pertaining to serial IO.
 */
/*
 *      serial.c
 * 该程序用于实现 rs232 的输入输出功能
 *      void rs_write(struct tty_struct *queue);
 *      void rs_init(void);
 * 以及与传输 IO 有关系的所有中断处理程序。
 */

#include <linux/tty.h>     // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <asm/system.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
#include <asm/io.h>        // io 头文件。定义硬件端口输入/输出宏汇编语句。

#define WAKEUP_CHARS (TTY_BUF_SIZE/4)   // 当写队列中含有 WAKEUP_CHARS 个字符时，就开始发送。

extern void rs1_interrupt(void);        // 串行口 1 的中断处理程序(rs_io.s, 34)。
extern void rs2_interrupt(void);        // 串行口 2 的中断处理程序(rs_io.s, 38)。

//// 初始化串行端口
// port: 串口 1 - 0x3F8，串口 2 - 0x2F8。
static void init(int port)
{
        outb_p(0x80,port+3);    /* set DLAB of line control reg */
                                /* 设置线路控制寄存器的 DLAB 位(位 7) */
        outb_p(0x30,port);      /* LS of divisor (48 -> 2400 bps */
                                /* 发送波特率因子低字节，0x30->2400bps */
        outb_p(0x00,port+1);    /* MS of divisor */
                                /* 发送波特率因子高字节，0x00 */
        outb_p(0x03,port+3);    /* reset DLAB */
                                /* 复位 DLAB 位，数据位为 8 位 */
        outb_p(0x0b,port+4);    /* set DTR,RTS, OUT_2 */
                                /* 设置 DTR，RTS，辅助用户输出 2 */
        outb_p(0x0d,port+1);    /* enable all intrs but writes */
                                /* 除了写(写保持空)以外，允许所有中断源中断 */
        (void)inb(port);        /* read data port to reset things (?) */
                                /* 读数据口，以进行复位操作(?) */
}

//// 初始化串行中断程序和串行接口。
void rs_init(void)
{
        set_intr_gate(0x24,rs1_interrupt);   // 设置串行口 1 的中断门向量(硬件 IRQ4 信号)。
        set_intr_gate(0x23,rs2_interrupt);   // 设置串行口 2 的中断门向量(硬件 IRQ3 信号)。
        init(tty_table[1].read_q.data);      // 初始化串行口 1(.data 是端口号)。
        init(tty_table[2].read_q.data);      // 初始化串行口 2。
        outb(inb_p(0x21)&0xE7,0x21);         // 允许主 8259A 芯片的 IRQ3，IRQ4 中断信号请求。
}

/*
 * This routine gets called when tty_write has put something into
```

```
 * the write_queue. It must check wheter the queue is empty, and
 * set the interrupt register accordingly
 *
 *      void _rs_write(struct tty_struct * tty);
 */
/*
 * 在 tty_write()已将数据放入输出(写)队列时会调用下面的子程序。必须首先
 * 检查写队列是否为空，并相应设置中断寄存器。
 */
//// 串行数据发送输出。
// 实际上只是开启串行发送保持寄存器已空中断标志，在 UART 将数据发送出去后允许发中断信号。
void rs_write(struct tty_struct * tty)
{
        cli();                          // 关中断。
// 如果写队列不空，则从 0x3f9(或 0x2f9) 首先读取中断允许寄存器内容，添上发送保持寄存器
// 中断允许标志（位 1）后，再写回该寄存器。
        if (!EMPTY(tty->write_q))
                outb(inb_p(tty->write_q.data+1)|0x02,tty->write_q.data+1);
        sti();                          // 开中断。
}

```

7.3.3 其它信息

7.3.3.1 异步串行通信芯片 UART

PC 微机的串行通信使用的异步串行通信芯片是 INS 8250 或 NS16450 兼容芯片，统称为 UART(通用异步接收发送器)。对 UART 的编程实际上是对其内部寄存器执行读写操作。因此可将 UART 看作是一组寄存器集合，包含发送、接收和控制三部分。UART 内部有 10 个寄存器，供 CPU 通过 IN/OUT 指令对其进行访问。这些寄存器的端口和用途见下表所示。其中端口 0x3f8-0x3fe 用于微机上 COM1 串行口，0x2f8-0x2fe 对应 COM2 端口。条件 DLAB(Divisor Latch Access Bit)是除数锁存访问位，是指线路控制寄存器的位 7。

表 7.1 UART 内部寄存器对应端口及用途

端口	读/写	条件	用途
0x3f8 (0x2f8)	写	DLAB=0	写发送保持寄存器。含有将发送的字符。
	读	DLAB=0	读接收缓存寄存器。含有收到的字符。
	读/写	DLAB=1	读/写波特率因子低字节（LSB)。
0x3f9 (0x2f9)	读/写	DLAB=1	读/写波特率因子高字节（MSB)。
	读/写	DLAB=0	读/写中断允许寄存器。 位 7-4 全 0 保留不用； 位 3=1 modem 状态中断允许； 位 2=1 接收器线路状态中断允许； 位 1=1 发送保持寄存器空中断允许； 位 0=1 已接收到数据中断允许。
0x3fa (0x2fa)	读		读中断标识寄存器。中断处理程序用以判断此次中断是 4 种中的那一种。 位 7-3 全 0（不用)； 位 2-1 确定中断的优先级；

			= 11 接收状态有错中断，优先级最高； = 10 已接收到数据中断，优先级第 2； = 01 发送保持寄存器空中断，优先级第 3； = 00 modem 状态改变中断，优先级第 4。 位 0=0 有待处理中断；=1 无中断。
0x3fb (0x2fb)	写		写线路控制寄存器。 位 7=1 除数锁存访问位(DLAB)。 0 接收器，发送保持或中断允许寄存器访问； 位 6=1 允许间断； 位 5=1 保持奇偶位； 位 4=1 偶校验；=0 奇校验； 位 3=1 允许奇偶校验；=0 无奇偶校验； 位 2=1 1 位停止位；=0 无停止位； 位 1-0 数据位长度： = 00 5 位数据位； = 01 6 位数据位； = 10 7 位数据位； = 11 8 位数据位。
0x3fc (0x2fc)	写		写 modem 控制寄存器。 位 7-5 全 0 保留； 位 4=1 芯片处于循环反馈诊断操作模式； 位 3=1 辅助用户指定输出 2，允许 INTRPT 到系统； 位 2=1 辅助用户指定输出 1，PC 机未用； 位 1=1 使请求发送 RTS 有效； 位 0=1 使数据终端就绪 DTR 有效。
0x3fd (0x2fd)	读		读线路状态寄存器。 位 7=0 保留； 位 6=1 发送移位寄存器为空； 位 5=1 发送保持寄存器为空，可以取字符发送； 位 4=1 接收到满足间断条件的位序列； 位 3=1 帧格式错误； 位 2=1 奇偶校验错误； 位 1=1 超越覆盖错误； 位 0=1 接收器数据准备好，系统可读取。
0x3fe (0x2fe)	读		读 modem 状态寄存器。δ 表示信号发生变化。 位 7=1 载波检测(CD)有效； 位 6=1 响铃指示(RI)有效； 位 5=1 数据设备就绪(DSR)有效； 位 4=1 清除发送（CTS）有效； 位 3=1 检测到 δ 载波； 位 2=1 检测到响铃信号边沿； 位 1=1 δ 数据设备就绪(DSR)； 位 0=1 δ 清除发送(CTS)。

7.4 rs_io.s 文件

7.4.1 功能描述

该汇编程序实现 rs232 串行通信中断处理程序。

7.4.2 代码注释

列表 7.4 linux/kernel/chr_drv/rs_io.s 程序

```
/*
 *  linux/kernel/rs_io.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      rs_io.s
 *
 * This module implements the rs232 io interrupts.
 */
/*
 * 该程序模块实现 rs232 输入输出中断处理程序。
 */

.text
.globl _rs1_interrupt,_rs2_interrupt

// size 是读写队列缓冲区的字节长度。
size    = 1024                          /* must be power of two !    必须是 2 的次方并且需
                                           and must match the value  与 tty_io.c 中的值匹配!
                                           in tty_io.c!!! */

/* these are the offsets into the read/write buffer structures */
/* 以下这些是读写缓冲结构中的偏移量 */
// 对应定义在 include/linux/tty.h 文件中 tty_queue 结构中各变量的偏移量。
rs_addr = 0                // 串行端口号字段偏移（端口号是 0x3f8 或 0x2f8）。
head = 4                   // 缓冲区中头指针字段偏移。
tail = 8                   // 缓冲区中尾指针字段偏移。
proc_list = 12             // 等待该缓冲的进程字段偏移。
buf = 16                   // 缓冲区字段偏移。

startup = 256              /* chars left in write queue when we restart it */
                           /* 当写队列里还剩 256 个字符空间(WAKEUP_CHARS)时，我们就可以写 */

/*
 * These are the actual interrupt routines. They look where
 * the interrupt is coming from, and take appropriate action.
 */
/*
 * 这些是实际的中断程序。程序首先检查中断的来源，然后执行相应
```

```
    * 的处理。
    */
33 .align 2
   //// 串行端口 1 中断处理程序入口点。
34 _rs1_interrupt:
35         pushl $_table_list+8   // tty 表中对应串口 1 的读写缓冲指针的地址入栈(tty_io.c，99)。
36         jmp rs_int
37 .align 2
   //// 串行端口 2 中断处理程序入口点。
38 _rs2_interrupt:
39         pushl $_table_list+16  // tty 表中对应串口 2 的读写缓冲队列指针的地址入栈。
40 rs_int:
41         pushl %edx
42         pushl %ecx
43         pushl %ebx
44         pushl %eax
45         push %es
46         push %ds               /* as this is an interrupt, we cannot */
47         pushl $0x10            /* know that bs is ok. Load it */
48         pop %ds                /* 由于这是一个中断程序，我们不知道 ds 是否正确，*/
49         pushl $0x10            /* 所以加载它们(让 ds、es 指向内核数据段 */
50         pop %es
51         movl 24(%esp),%edx     // 将缓冲队列指针地址存入 edx 寄存器，
                                  // 也即 35 或 39 行上最先压入堆栈的地址。
52         movl (%edx),%edx       // 取读队列指针(地址)➔edx。
53         movl rs_addr(%edx),%edx // 取串口 1 的端口号➔edx。
54         addl $2,%edx           /* interrupt ident. reg */  /* edx 指向中断标识寄存器 */
55 rep_int:                       // 中断标识寄存器端口是 0x3fa（0x2fa），参见上节列表后信息。
56         xorl %eax,%eax         // eax 清零。
57         inb %dx,%al            // 取中断标识字节，用以判断中断来源(有 4 种中断情况)。
58         testb $1,%al           // 首先判断有无待处理的中断(位 0=1 无中断；=0 有中断)。
59         jne end                // 若无待处理中断，则跳转至退出处理处 end。
60         cmpb $6,%al            /* this shouldn't happen, but ... */ /* 这不会发生，但是…*/
61         ja end                 // al 值>6? 是则跳转至 end（没有这种状态）。
62         movl 24(%esp),%ecx     // 再取缓冲队列指针地址➔ecx。
63         pushl %edx             // 将端口号 0x3fa(0x2fa)入栈。
64         subl $2,%edx           // 0x3f8(0x2f8)。
65         call jmp_table(,%eax,2) /* NOTE! not *4, bit0 is 0 already */ /* 不乘 4，位 0 已是 0*/
   // 上面语句是指，当有待处理中断时，al 中位 0=0，位 2-1 是中断类型，因此相当于已经将中断类型
   // 乘了 2，这里再乘 2，得到跳转表对应各中断类型地址，并跳转到那里去作相应处理。
66         popl %edx              // 弹出中断标识寄存器端口号 0x3fa（或 0x2fa）。
67         jmp rep_int            // 跳转，继续判断有无待处理中断并继续处理。
68 end:    movb $0x20,%al         // 向中断控制器发送结束中断指令 EOI。
69         outb %al,$0x20         /* EOI */
70         pop %ds
71         pop %es
72         popl %eax
73         popl %ebx
74         popl %ecx
75         popl %edx
76         addl $4,%esp           # jump over _table_list entry # 丢弃缓冲队列指针地址。
77         iret
78
```

```
    // 各中断类型处理程序地址跳转表，共有 4 种中断来源：
    // modem 状态变化中断，写字符中断，读字符中断，线路状态有问题中断。
jmp_table:
        .long modem_status,write_char,read_char,line_status

.align 2
modem_status:
        addl $6,%edx            /* clear intr by reading modem status reg */
        inb %dx,%al             /* 通过读 modem 状态寄存器进行复位(0x3fe) */
        ret

.align 2
line_status:
        addl $5,%edx            /* clear intr by reading line status reg. */
        inb %dx,%al             /* 通过读线路状态寄存器进行复位(0x3fd) */
        ret

.align 2
read_char:
        inb %dx,%al             /* 读取字符➔al。
        movl %ecx,%edx          /* 当前串口缓冲队列指针地址➔edx。
        subl $_table_list,%edx  // 缓冲队列指针表首址 - 当前串口队列指针地址➔edx，
        shrl $3,%edx            // 差值/8。对于串口 1 是 1，对于串口 2 是 2。
        movl (%ecx),%ecx                # read-queue # 取读缓冲队列结构地址➔ecx。
        movl head(%ecx),%ebx    // 取读队列中缓冲头指针➔ebx。
        movb %al,buf(%ecx,%ebx) // 将字符放在缓冲区中头指针所指的位置。
        incl %ebx               // 将头指针前移一字节。
        andl $size-1,%ebx       // 用缓冲区大小对头指针进行模操作。指针不能超过缓冲区大小。
        cmpl tail(%ecx),%ebx    // 缓冲区头指针与尾指针比较。
        je 1f                   // 若相等，表示缓冲区满，跳转到标号 1 处。
        movl %ebx,head(%ecx)    // 保存修改过的头指针。
1:      pushl %edx              // 将串口号压入堆栈(1- 串口 1，2 - 串口 2)，作为参数，
        call _do_tty_interrupt  // 调用 tty 中断处理 C 函数（。
        addl $4,%esp            // 丢弃入栈参数，并返回。
        ret

.align 2
write_char:
        movl 4(%ecx),%ecx               # write-queue # 取写缓冲队列结构地址➔ecx。
        movl head(%ecx),%ebx            // 取写队列头指针➔ebx。
        subl tail(%ecx),%ebx            // 头指针 - 尾指针 = 队列中字符数。
        andl $size-1,%ebx               # nr chars in queue  # 对指针取模运算。
        je write_buffer_empty          // 如果头指针 = 尾指针，说明写队列无字符，跳转处理。
        cmpl $startup,%ebx              // 队列中字符数超过 256 个？
        ja 1f                           // 超过，则跳转处理。
        movl proc_list(%ecx),%ebx       # wake up sleeping process # 唤醒等待的进程。
                                        // 取等待该队列的进程的指针，并判断是否为空。
        testl %ebx,%ebx                 # is there any?  # 有等待的进程吗？
        je 1f                           // 是空的，则向前跳转到标号 1 处。
        movl $0,(%ebx)                  // 否则将进程置为可运行状态(唤醒进程)。。
1:      movl tail(%ecx),%ebx            // 取尾指针。
        movb buf(%ecx,%ebx),%al         // 从缓冲中尾指针处取一字符➔al。
        outb %al,%dx                    // 向端口 0x3f8(0x2f8)送出到保持寄存器中。
```

```
        incl %ebx                       // 尾指针前移。
        andl $size-1,%ebx               // 尾指针若到缓冲区末端，则折回。
        movl %ebx,tail(%ecx)            // 保存已修改过的尾指针。
        cmpl head(%ecx),%ebx            // 尾指针与头指针比较，
        je write_buffer_empty           // 若相等，表示队列已空，则跳转。
        ret
.align 2
write_buffer_empty:
        movl proc_list(%ecx),%ebx       # wake up sleeping process # 唤醒等待的进程。
                                        // 取等待该队列的进程的指针，并判断是否为空。
        testl %ebx,%ebx                 # is there any?  # 有等待的进程吗？
        je 1f                           # 无，则向前跳转到标号 1 处。
        movl $0,(%ebx)                  # 否则将进程置为可运行状态(唤醒进程)。
1:      incl %edx                       # 指向端口 0x3f9(0x2f9)。
        inb %dx,%al                     # 读取中断允许寄存器。
        jmp 1f                          # 稍作延迟。
1:      jmp 1f
1:      andb $0xd,%al               /* disable transmit interrupt */
                                    /* 屏蔽发送保持寄存器空中断（位 1） */
        outb %al,%dx                    // 写入 0x3f9(0x2f9)。
        ret
```

7.5 tty_io.c 文件

7.5.1 功能描述

7.5.2 代码注释

列表 7.5 linux/kernel/chr_drv/tty_io.c 程序

```
/*
 *  linux/kernel/tty_io.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * 'tty_io.c' gives an orthogonal feeling to tty's, be they consoles
 * or rs-channels. It also implements echoing, cooked mode etc.
 *
 * Kill-line thanks to John T Kohl.
 */
/*
* 'tty_io.c'给 tty 一种非相关的感觉，是控制台还是串行通道。该程序同样
* 实现了回显、规范(熟)模式等。
*
* Kill-line，谢谢 John T Kahl。
*/
#include <ctype.h>          // 字符类型头文件。定义了一些有关字符类型判断和转换的宏。
```

```
#include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
#include <signal.h>         // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。

// 下面给出相应信号在信号位图中的对应比特位。
#define ALRMMASK (1<<(SIGALRM-1))      // 警告(alarm)信号屏蔽位。
#define KILLMASK (1<<(SIGKILL-1))      // 终止(kill)信号屏蔽位。
#define INTMASK (1<<(SIGINT-1))        // 键盘中断(int)信号屏蔽位。
#define QUITMASK (1<<(SIGQUIT-1))      // 键盘退出(quit)信号屏蔽位。
#define TSTPMASK (1<<(SIGTSTP-1))      // tty 发出的停止进程(tty stop)信号屏蔽位。

#include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                            // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/tty.h>      // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
#include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
#include <asm/system.h>     // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

#define _L_FLAG(tty,f)   ((tty)->termios.c_lflag & f)   // 取 termios 结构中的本地模式标志。
#define _I_FLAG(tty,f)   ((tty)->termios.c_iflag & f)   // 取 termios 结构中的输入模式标志。
#define _O_FLAG(tty,f)   ((tty)->termios.c_oflag & f)   // 取 termios 结构中的输出模式标志。

// 取 termios 结构中本地模式标志集中的一个标志位。
#define L_CANON(tty)     _L_FLAG((tty),ICANON)     // 取本地模式标志集中规范（熟）模式标志位。
#define L_ISIG(tty)      _L_FLAG((tty),ISIG)       // 取信号标志位。
#define L_ECHO(tty)      _L_FLAG((tty),ECHO)       // 取回显字符标志位。
#define L_ECHOE(tty)     _L_FLAG((tty),ECHOE)      // 规范模式时，取回显擦出标志位。
#define L_ECHOK(tty)     _L_FLAG((tty),ECHOK)      // 规范模式时，取 KILL 擦除当前行标志位。
#define L_ECHOCTL(tty)   _L_FLAG((tty),ECHOCTL)    // 取回显控制字符标志位。
#define L_ECHOKE(tty)    _L_FLAG((tty),ECHOKE)     // 规范模式时，取 KILL 擦除行并回显标志位。

// 取 termios 结构中输入模式标志中的一个标志位。
#define I_UCLC(tty)      _I_FLAG((tty),IUCLC)      // 取输入模式标志集中大写到小写转换标志位。
#define I_NLCR(tty)      _I_FLAG((tty),INLCR)      // 取换行符 NL 转回车符 CR 标志位。
#define I_CRNL(tty)      _I_FLAG((tty),ICRNL)      // 取回车符 CR 转换行符 NL 标志位。
#define I_NOCR(tty)      _I_FLAG((tty),IGNCR)      // 取忽略回车符 CR 标志位。

// 取 termios 结构中输出模式标志中的一个标志位。
#define O_POST(tty)      _O_FLAG((tty),OPOST)      // 取输出模式标志集中执行输出处理标志。
#define O_NLCR(tty)      _O_FLAG((tty),ONLCR)      // 取换行符 NL 转回车换行符 CR-NL 标志。
#define O_CRNL(tty)      _O_FLAG((tty),OCRNL)      // 取回车符 CR 转换行符 NL 标志。
#define O_NLRET(tty)     _O_FLAG((tty),ONLRET)     // 取换行符 NL 执行回车功能的标志。
#define O_LCUC(tty)      _O_FLAG((tty),OLCUC)      // 取小写转大写字符标志。

// tty 数据结构的 tty_table 数组。其中包含三个初始化项数据，分别对应控制台、串口终端 1 和
// 串口终端 2 的初始化数据。
struct tty_struct tty_table[] = {
        {
                {ICRNL,         /* change incoming CR to NL */   /* 将输入的 CR 转换为 NL */
                OPOST|ONLCR,    /* change outgoing NL to CRNL */ /* 将输出的 NL 转 CRNL */
                0,              // 控制模式标志初始化为 0。
                ISIG | ICANON | ECHO | ECHOCTL | ECHOKE,   // 本地模式标志。
                0,              /* console termio */       // 控制台 termio，不用线路规程。
                INIT_C_CC},                                // 控制字符数组。
                0,              /* initial pgrp */         // 所属初始进程组。
```

```
                0,              /* initial stopped */       // 初始停止标志。
                con_write,                                  // tty 写函数指针。
                {0,0,0,0,""},   /* console read-queue */    // tty 控制台读队列。
                {0,0,0,0,""},   /* console write-queue */   // tty 控制台写队列。
                {0,0,0,0,""}    /* console secondary queue */  // tty 控制台辅助(第二)队列。
        },{
                {0, /* no translation */        // 输入模式标志。0，无须转换。
                0,  /* no translation */        // 输出模式标志。0，无须转换。
                B2400 | CS8,                    // 控制模式标志。波特率 2400bps，8 位数据位。
                0,                              // 本地模式标志 0。
                0,                              // 线路规程 0。
                INIT_C_CC},                     // 控制字符数组。
                0,                              // 所属初始进程组。
                0,                              // 初始停止标志。
                rs_write,                       // 串口 1 tty 写函数指针。
                {0x3f8,0,0,0,""},               /* rs 1 */  // 串行终端 1 读缓冲队列。
                {0x3f8,0,0,0,""},               // 串行终端 1 写缓冲队列。
                {0,0,0,0,""}                    // 串行终端 1 辅助缓冲队列。
        },{
                {0, /* no translation */        // 输入模式标志。0，无须转换。
                0,  /* no translation */        // 输出模式标志。0，无须转换。
                B2400 | CS8,                    // 控制模式标志。波特率 2400bps，8 位数据位。
                0,                              // 本地模式标志 0。
                0,                              // 线路规程 0。
                INIT_C_CC},                     // 控制字符数组。
                0,                              // 所属初始进程组。
                0,                              // 初始停止标志。
                rs_write,                       // 串口 2 tty 写函数指针。
                {0x2f8,0,0,0,""},                 /* rs 2 */  // 串行终端 2 读缓冲队列。
                {0x2f8,0,0,0,""},               // 串行终端 2 写缓冲队列。
                {0,0,0,0,""}                    // 串行终端 2 辅助缓冲队列。
        }
};

/*
 * these are the tables used by the machine code handlers.
 * you can implement pseudo-tty's or something by changing
 * them. Currently not done.
 */
/*
 * 下面是汇编程序使用的缓冲队列地址表。通过修改你可以实现
 * 伪 tty 终端或其它终端类型。目前还没有这样做。
 */
// tty 缓冲队列地址表。rs_io.s 汇编程序使用，用于取得读写缓冲队列地址。
struct tty_queue * table_list[]={
        &tty_table[0].read_q, &tty_table[0].write_q,  // 控制台终端读、写缓冲队列地址。
        &tty_table[1].read_q, &tty_table[1].write_q,  // 串行口 1 终端读、写缓冲队列地址。
        &tty_table[2].read_q, &tty_table[2].write_q   // 串行口 2 终端读、写缓冲队列地址。
        };

//// tty 终端初始化函数。
// 初始化串口终端和控制台终端。
void tty_init(void)
```

```
{
        rs_init();                    // 初始化串行中断程序和串行接口 1 和 2。(serial.c, 37)
        con_init();                   // 初始化控制台终端。(console.c, 617)
}

//// tty 键盘终端字符处理函数。
// 参数：tty - 相应 tty 终端结构指针；mask - 信号屏蔽位。
void tty_intr(struct tty_struct * tty, int mask)
{
        int i;

// 如果 tty 所属组号小于等于 0，则退出。
        if (tty->pgrp <= 0)
                return;
// 扫描任务数组，向 tty 相应组的所有任务发送指定的信号。
        for (i=0;i<NR_TASKS;i++)
// 如果该项任务指针不为空，并且其组号等于 tty 组号，则设置该任务指定的信号 mask。
                if (task[i] && task[i]->pgrp==tty->pgrp)
                        task[i]->signal |= mask;
}

//// 如果队列缓冲区空则让进程进入可中断的睡眠状态。
// 参数：queue - 指定队列的指针。
// 进程在取队列缓冲区中字符时调用此函数。
static void sleep_if_empty(struct tty_queue * queue)
{
        cli();                        // 关中断。
// 若当前进程没有信号要处理并且指定的队列缓冲区空，则让进程进入可中断睡眠状态，并让
// 队列的进程等待指针指向该进程。
        while (!current->signal && EMPTY(*queue))
                interruptible_sleep_on(&queue->proc_list);
        sti();                        // 开中断。
}

//// 若队列缓冲区满则让进程进入可中断的睡眠状态。
// 参数：queue - 指定队列的指针。
// 进程在往队列缓冲区中写入时调用此函数。
static void sleep_if_full(struct tty_queue * queue)
{
// 若队列缓冲区不满，则返回退出。
        if (!FULL(*queue))
                return;
        cli();                        // 关中断。
// 如果进程没有信号需要处理并且队列缓冲区中空闲剩余区长度<128，则让进程进入可中断睡眠状态，
// 并让该队列的进程等待指针指向该进程。
        while (!current->signal && LEFT(*queue)<128)
                interruptible_sleep_on(&queue->proc_list);
        sti();                        // 开中断。
}

//// 等待按键。
// 如果控制台的读队列缓冲区空则让进程进入可中断的睡眠状态。
void wait_for_keypress(void)
```

```
{
        sleep_if_empty(&tty_table[0].secondary);
}

//// 复制成规范模式字符序列。
// 将指定 tty 终端队列缓冲区中的字符复制成规范(熟)模式字符并存放在辅助队列(规范模式队列)中。
// 参数：tty - 指定终端的 tty 结构。
void copy_to_cooked(struct tty_struct * tty)
{
        signed char c;

// 如果 tty 的读队列缓冲区不空并且辅助队列缓冲区为空，则循环执行下列代码。
        while (!EMPTY(tty->read_q) && !FULL(tty->secondary)) {
// 从队列尾处取一字符到 c，并前移尾指针。
                GETCH(tty->read_q,c);
// 下面对输入字符，利用输入模式标志集进行处理。
// 如果该字符是回车符 CR(13)，则：若回车转换行标志 CRNL 置位则将该字符转换为换行符 NL(10)；
// 否则若忽略回车标志 NOCR 置位，则忽略该字符，继续处理其它字符。
                if (c==13)
                        if (I_CRNL(tty))
                                c=10;
                        else if (I_NOCR(tty))
                                continue;
                        else ;
// 如果该字符是换行符 NL(10)并且换行转回车标志 NLCR 置位，则将其转换为回车符 CR(13)。
                else if (c==10 && I_NLCR(tty))
                        c=13;
// 如果大写转小写标志 UCLC 置位，则将该字符转换为小写字符。
                if (I_UCLC(tty))
                        c=tolower(c);
// 如果本地模式标志集中规范（熟）模式标志 CANON 置位，则进行以下处理。
                if (L_CANON(tty)) {
// 如果该字符是键盘终止控制字符 KILL(^U)，则进行删除输入行处理。
                        if (c==KILL_CHAR(tty)) {
                                /* deal with killing the input line */ /* 删除输入行处理 */
// 如果 tty 辅助队列不空，或者辅助队列中最后一个字符是换行 NL(10)，或者该字符是文件结束字符
// (^D)，则循环执行下列代码。
                                while(!(EMPTY(tty->secondary) ||
                                        (c=LAST(tty->secondary))==10 ||
                                        c==EOF_CHAR(tty))) {
// 如果本地回显标志 ECHO 置位，那么：若字符是控制字符(值<32)，则往 tty 的写队列中放入擦除
// 字符 ERASE。再放入一个擦除字符 ERASE，并且调用该 tty 的写函数。
                                        if (L_ECHO(tty)) {
                                                if (c<32)
                                                        PUTCH(127,tty->write_q);
                                                PUTCH(127,tty->write_q);
                                                tty->write(tty);
                                        }
// 将 tty 辅助队列头指针后退 1 字节。
                                        DEC(tty->secondary.head);
                                }
                                continue;     // 继续读取并处理其它字符。
                        }
```

```
// 如果该字符是删除控制字符 ERASE(^H)，那么：
			if (c==ERASE_CHAR(tty)) {
// 若 tty 的辅助队列为空，或者其最后一个字符是换行符 NL(10)，或者是文件结束符，继续处理
// 其它字符。
				if (EMPTY(tty->secondary) ||
				    (c=LAST(tty->secondary))==10 ||
				    c==EOF_CHAR(tty))
					continue;
// 如果本地回显标志 ECHO 置位，那么：若字符是控制字符(值<32)，则往 tty 的写队列中放入擦除
// 字符 ERASE。再放入一个擦除字符 ERASE，并且调用该 tty 的写函数。
				if (L_ECHO(tty)) {
					if (c<32)
						PUTCH(127,tty->write_q);
					PUTCH(127,tty->write_q);
					tty->write(tty);
				}
// 将 tty 辅助队列头指针后退 1 字节，继续处理其它字符。
				DEC(tty->secondary.head);
				continue;
			}
//如果该字符是停止字符(^S)，则置 tty 停止标志，继续处理其它字符。
			if (c==STOP_CHAR(tty)) {
				tty->stopped=1;
				continue;
			}
// 如果该字符是停止字符(^Q)，则复位 tty 停止标志，继续处理其它字符。
			if (c==START_CHAR(tty)) {
				tty->stopped=0;
				continue;
			}
		}
// 若输入模式标志集中 ISIG 标志置位，则在收到 INTR、QUIT、SUSP 或 DSUSP 字符时，需要为进程
// 产生相应的信号。
		if (L_ISIG(tty)) {
// 如果该字符是键盘中断符(^C)，则向当前进程发送键盘中断信号，并继续处理下一字符。
			if (c==INTR_CHAR(tty)) {
				tty_intr(tty,INTMASK);
				continue;
			}
// 如果该字符是键盘中断符(^\)，则向当前进程发送键盘退出信号，并继续处理下一字符。
			if (c==QUIT_CHAR(tty)) {
				tty_intr(tty,QUITMASK);
				continue;
			}
		}
// 如果该字符是换行符 NL(10)，或者是文件结束符 EOF(^D)，辅助缓冲队列字符数加 1。[??]
		if (c==10 || c==EOF_CHAR(tty))
			tty->secondary.data++;
// 如果本地模式标志集中回显标志 ECHO 置位，那么，如果字符是换行符 NL(10)，则将换行符 NL(10)
// 和回车符 CR(13)放入 tty 写队列缓冲区中；如果字符是控制字符(字符值<32)并且回显控制字符标志
// ECHOCTL 置位，则将字符'^'和字符 c+64 放入 tty 写队列中(也即会显示^C、^H 等)；否则将该字符
// 直接放入 tty 写缓冲队列中。最后调用该 tty 的写操作函数。
		if (L_ECHO(tty)) {
```

```
                    if (c==10) {
                        PUTCH(10,tty->write_q);
                        PUTCH(13,tty->write_q);
                    } else if (c<32) {
                        if (L_ECHOCTL(tty)) {
                            PUTCH('^',tty->write_q);
                            PUTCH(c+64,tty->write_q);
                        }
                    } else
                        PUTCH(c,tty->write_q);
                    tty->write(tty);
                }
// 将该字符放入辅助队列中。
                PUTCH(c,tty->secondary);
        }
// 唤醒等待该辅助缓冲队列的进程（如果有的话）。
        wake_up(&tty->secondary.proc_list);
}

//// tty 读函数。
// 参数：channel - 子设备号；buf - 缓冲区指针；nr - 欲读字节数。
// 返回已读字节数。
int tty_read(unsigned channel, char * buf, int nr)
{
        struct tty_struct * tty;
        char c, * b=buf;
        int minimum,time,flag=0;
        long oldalarm;

// 本版本 linux 内核的终端只有 3 个子设备，分别是控制台(0)、串口终端 1(1)和串口终端 2(2)。
// 所以任何大于 2 的子设备号都是非法的。写的字节数当然也不能小于 0 的。
        if (channel>2 || nr<0) return -1;
// tty 指针指向子设备号对应 ttb_table 表中的 tty 结构。
        tty = &tty_table[channel];
// 下面首先保存进程原定时值，然后根据控制字符 VTIME 和 VMIN 设置读字符操作的超时定时值。
// 在非规范模式下，这两个值是超时定时值。MIN 表示为了满足读操作，需要读取的最少字符数。
// TIME 是一个十分之一秒计数的计时值。
// 首先取进程中的(报警)定时值(滴答数)。
        oldalarm = current->alarm;
// 并设置读操作超时定时值 time 和需要最少读取的字符个数 minimum。
        time = 10L*tty->termios.c_cc[VTIME];
        minimum = tty->termios.c_cc[VMIN];
// 如果设置了读超时定时值 time 但没有设置最少读取个数 minimum，那么在读到至少一个字符或者
// 定时超时后读操作将立刻返回。所以这里置 minimum=1。
        if (time && !minimum) {
                minimum=1;
// 如果进程原定时值是 0 或者 time+当前系统时间值小于进程原定时值的话，则置重新设置进程定时
// 值为 time+当前系统时间，并置 flag 标志。
                if (flag=(!oldalarm || time+jiffies<oldalarm))
                    current->alarm = time+jiffies;
        }
// 如果设置的最少读取字符数>欲读的字符数，则令其等于此次欲读取的字符数。
        if (minimum>nr)
```

```
                minimum=nr;
// 当欲读的字节数>0，则循环执行以下操作。
        while (nr>0) {
// 如果 flag 不为 0(即进程原定时值是 0 或者 time+当前系统时间值小于进程原定时值)并且进程有定
// 时信号 SIGALRM，则复位进程的定时信号并中断循环。
                if (flag && (current->signal & ALRMMASK)) {
                        current->signal &= ~ALRMMASK;
                        break;
                }
// 如果当前进程有信号要处理，则退出，返回 0。
                if (current->signal)
                        break;
// 如果辅助缓冲队列(规范模式队列)为空，或者设置了规范模式标志并且辅助队列中字符数为 0 以及
// 辅助模式缓冲队列空闲空间>20，则进入可中断睡眠状态，返回后继续处理。
                if (EMPTY(tty->secondary) || (L_CANON(tty) &&
                !tty->secondary.data && LEFT(tty->secondary)>20)) {
                        sleep_if_empty(&tty->secondary);
                        continue;
                }
// 执行以下操作，直到 nr=0 或者辅助缓冲队列为空。
                do {
// 取辅助缓冲队列字符 c。
                        GETCH(tty->secondary,c);
// 如果该字符是文件结束符(^D)或者是换行符 NL(10)，则辅助缓冲队列字符数减 1。
                        if (c==EOF_CHAR(tty) || c==10)
                                tty->secondary.data--;
// 如果该字符是文件结束符(^D)并且规范模式标志置位，则返回已读字符数，并退出。
                        if (c==EOF_CHAR(tty) && L_CANON(tty))
                                return (b-buf);
// 否则将该字符放入用户数据段缓冲区 buf 中，欲读字符数减 1，如果欲读字符数已为 0，则中断循环。
                        else {
                                put_fs_byte(c,b++);
                                if (!--nr)
                                        break;
                        }
                } while (nr>0 && !EMPTY(tty->secondary));
// 如果超时定时值 time 不为 0 并且规范模式标志没有置位(非规范模式)，那么：
                if (time && !L_CANON(tty))
// 如果进程原定时值是 0 或者 time+当前系统时间值小于进程原定时值的话，则置重新设置进程定时值
// 为 time+当前系统时间，并置 flag 标志。否则让进程的定时值等于进程原定时值。
                        if (flag=(!oldalarm || time+jiffies<oldalarm))
                                current->alarm = time+jiffies;
                        else
                                current->alarm = oldalarm;
// 如果规范模式标志置位，那么若没有读到 1 个字符则中断循环。否则若已读取数大于或等于最少要
// 求读取的字符数，则也中断循环。
                if (L_CANON(tty)) {
                        if (b-buf)
                                break;
                } else if (b-buf >= minimum)
                        break;
        }
// 让进程的定时值等于进程原定时值。
```

```
        current->alarm = oldalarm;
// 如果进程有信号并且没有读取任何字符，则返回出错号（超时）。
        if (current->signal && !(b-buf))
                return -EINTR;
        return (b-buf);                // 返回已读取的字符数。
}

//// tty 写函数。
// 参数：channel - 子设备号；buf - 缓冲区指针；nr - 写字节数。
// 返回已写字节数。
int tty_write(unsigned channel, char * buf, int nr)
{
        static cr_flag=0;
        struct tty_struct * tty;
        char c, *b=buf;

// 本版本 linux 内核的终端只有 3 个子设备，分别是控制台(0)、串口终端 1(1)和串口终端 2(2)。
// 所以任何大于 2 的子设备号都是非法的。写的字节数当然也不能小于 0 的。
        if (channel>2 || nr<0) return -1;
// tty 指针指向子设备号对应 ttb_table 表中的 tty 结构。
        tty = channel + tty_table;
// 字符设备是一个一个字符进行处理的，所以这里对于 nr 大于 0 时对每个字符进行循环处理。
        while (nr>0) {
// 如果此时 tty 的写队列已满，则当前进程进入可中断的睡眠状态。
                sleep_if_full(&tty->write_q);
// 如果当前进程有信号要处理，则退出，返回 0。
                if (current->signal)
                        break;
// 当要写的字节数>0 并且 tty 的写队列不满时，循环执行以下操作。
                while (nr>0 && !FULL(tty->write_q)) {
// 从用户数据段内存中取一字节 c。
                        c=get_fs_byte(b);
// 如果终端输出模式标志集中执行输出处理标志 OPOST 置位，则执行下列输出时处理过程。
                        if (O_POST(tty)) {
// 如果该字符是回车符'\r'(CR，13)并且回车符转换行符标志 OCRNL 置位，则将该字符换成换行符
// '\n'(NL，10)；否则如果该字符是换行符'\n'(NL，10)并且换行转回车功能标志 ONLRET 置位的话，
// 则将该字符换成回车符'\r'(CR，13)。
                                if (c=='\r' && O_CRNL(tty))
                                        c='\n';
                                else if (c=='\n' && O_NLRET(tty))
                                        c='\r';
// 如果该字符是换行符'\n'并且回车标志 cr_flag 没有置位，换行转回车-换行标志 ONLCR 置位的话，
// 则将 cr_flag 置位，并将一回车符放入写队列中。然后继续处理下一个字符。
                                if (c=='\n' && !cr_flag && O_NLCR(tty)) {
                                        cr_flag = 1;
                                        PUTCH(13,tty->write_q);
                                        continue;
                                }
// 如果小写转大写标志 OLCUC 置位的话，就将该字符转成大写字符。
                                if (O_LCUC(tty))
                                        c=toupper(c);
                        }
// 用户数据缓冲指针 b 前进 1 字节；欲写字节数减 1 字节；复位 cr_flag 标志，并将该字节放入 tty
```

```
// 写队列中。
                                b++; nr--;
                                cr_flag = 0;
                                PUTCH(c,tty->write_q);
                        }
// 若字节全部写完，或者写队列已满，则程序执行到这里。调用对应 tty 的写函数，若还有字节要写，
// 则等待写队列不满，所以调用调度程序，先去执行其它任务。
                        tty->write(tty);
                        if (nr>0)
                                schedule();
                }
                return (b-buf);                 // 返回写入的字节数。
}

/*
 * Jeh, sometimes I really like the 386.
 * This routine is called from an interrupt,
 * and there should be absolutely no problem
 * with sleeping even in an interrupt (I hope).
 * Of course, if somebody proves me wrong, I'll
 * hate intel for all time :-). We'll have to
 * be careful and see to reinstating the interrupt
 * chips before calling this, though.
 *
 * I don't think we sleep here under normal circumstances
 * anyway, which is good, as the task sleeping might be
 * totally innocent.
 */
/*
 * 呵，有时我是真得很喜欢 386。该子程序是从一个中断处理程序中调用的，即使在
 * 中断处理程序中睡眠也应该绝对没有问题(我希望如此)。当然，如果有人证明我是
 * 错的，那么我将憎恨 intel 一辈子☺。但是我们必须小心，在调用该子程序之前需
 * 要恢复中断。
 *
 * 我不认为在通常环境下会处在这里睡眠，这样很好，因为任务睡眠是完全任意的。
 */
//// tty 中断处理调用函数 - 执行 tty 中断处理。
// 参数：tty - 指定的 tty 终端号（0，1 或 2）。
// 将指定 tty 终端队列缓冲区中的字符复制成规范(熟)模式字符并存放在辅助队列(规范模式队列)中。
// 在串口读字符中断(rs_io.s, 109)和键盘中断(kerboard.S, 69)中调用。
void do_tty_interrupt(int tty)
{
        copy_to_cooked(tty_table+tty);
}

//// 字符设备初始化函数。空，为以后扩展做准备。
void chr_dev_init(void)
{
}
```

7.5.3 其它信息

7.5.3.1 控制字符 VTIME、VMIN

在非规范模式下，这两个值是超时定时值。MIN 表示为了满足读操作，需要读取的最少字符数。TIME 是一个十分之一秒计数的计时值。当这两个都设置的话，读操作将等待，直到至少读到一个字符，然后在以读取 MIN 个字符或者时间 TIME 在读取最后一个字符后超时。如果仅设置了 MIN，那么在读取 MIN 个字符之前读操作将不返回。如果仅设置了 TIME，那么在读到至少一个字符或者定时超时后读操作将立刻返回。如果两个都没有设置，则读操作将立刻返回，仅给出目前已读的字节数。详细说明参见 termios.h 文件。

7.6 tty_ioctl.c 文件

7.6.1 功能描述

7.6.2 代码注释

列表 7.6 linux/kernel/chr_drv/tty_ioctl.c 程序

```
/*
 *  linux/kernel/chr_drv/tty_ioctl.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
#include <termios.h>        // 终端输入输出函数头文件。主要定义控制异步通信口的终端接口。

#include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                            // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
#include <linux/tty.h>      // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。

#include <asm/io.h>         // io 头文件。定义硬件端口输入/输出宏汇编语句。
#include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
#include <asm/system.h>     // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

// 这是波特率因子数组（或称为除数数组）。波特率与波特率因子的对应关系参见列表后的说明。
static unsigned short quotient[] = {
        0, 2304, 1536, 1047, 857,
        768, 576, 384, 192, 96,
        64, 48, 24, 12, 6, 3
};

//// 修改传输速率。
// 参数：tty - 终端对应的 tty 数据结构。
// 在除数锁存标志 DLAB(线路控制寄存器位 7)置位情况下，通过端口 0x3f8 和 0x3f9 向 UART 分别写入
// 波特率因子低字节和高字节。
static void change_speed(struct tty_struct * tty)
```

```
{
        unsigned short port,quot;

// 对于串口终端，其 tty 结构的读缓冲队列 data 字段存放的是串行端口号(0x3f8 或 0x2f8)。
        if (!(port = tty->read_q.data))
                return;
// 从 tty 的 termios 结构控制模式标志集中取得设置的波特率索引号，据此从波特率因子数组中取得
// 对应的波特率因子值。CBAUD 是控制模式标志集中波特率位屏蔽码。
        quot = quotient[tty->termios.c_cflag & CBAUD];
        cli();                          // 关中断。
        outb_p(0x80,port+3);            /* set DLAB */  // 首先设置除数锁定标志 DLAB。
        outb_p(quot & 0xff,port);       /* LS of divisor */  // 输出因子低字节。
        outb_p(quot >> 8,port+1);       /* MS of divisor */  // 输出因子高字节。
        outb(0x03,port+3);              /* reset DLAB */     // 复位 DLAB。
        sti();                          // 开中断。
}

//// 刷新 tty 缓冲队列。
// 参数：gueue - 指定的缓冲队列指针。
// 令缓冲队列的头指针等于尾指针，从而达到清空缓冲区(零字符)的目的。
static void flush(struct tty_queue * queue)
{
        cli();
        queue->head = queue->tail;
        sti();
}

//// 等待字符发送出去。
static void wait_until_sent(struct tty_struct * tty)
{
        /* do nothing - not implemented */  /* 什么都没做 - 还未实现 */
}

//// 发送 BREAK 控制符。
static void send_break(struct tty_struct * tty)
{
        /* do nothing - not implemented */  /* 什么都没做 - 还未实现 */
}

//// 取终端 termios 结构信息。
// 参数：tty - 指定终端的 tty 结构指针；termios - 用户数据区 termios 结构缓冲区指针。
// 返回 0 。
static int get_termios(struct tty_struct * tty, struct termios * termios)
{
        int i;

// 首先验证一下用户的缓冲区指针所指内存区是否足够，如不够则分配内存。
        verify_area(termios, sizeof (*termios));
// 复制指定 tty 结构中的 termios 结构信息到用户 termios 结构缓冲区。
        for (i=0 ; i< (sizeof (*termios)) ; i++)
                put_fs_byte( ((char *)&tty->termios)[i] , i+(char *)termios );
        return 0;
}
```

```
//// 设置终端 termios 结构信息。
// 参数：tty - 指定终端的 tty 结构指针；termios - 用户数据区 termios 结构指针。
// 返回 0 。
static int set_termios(struct tty_struct * tty, struct termios * termios)
{
        int i;

// 首先复制用户数据区中 termios 结构信息到指定 tty 结构中。
        for (i=0 ; i< (sizeof (*termios)) ; i++)
                ((char *)&tty->termios)[i]=get_fs_byte(i+(char *)termios);
// 用户有可能已修改了 tty 的串行口传输波特率，所以根据 termios 结构中的控制模式标志 c_cflag
// 修改串行芯片 UART 的传输波特率。
        change_speed(tty);
        return 0;
}

//// 读取 termio 结构中的信息。
// 参数：tty - 指定终端的 tty 结构指针；termio - 用户数据区 termio 结构缓冲区指针。
// 返回 0。
static int get_termio(struct tty_struct * tty, struct termio * termio)
{
        int i;
        struct termio tmp_termio;

// 首先验证一下用户的缓冲区指针所指内存区是否足够，如不够则分配内存。
        verify_area(termio, sizeof (*termio));
// 将 termios 结构的信息复制到 termio 结构中。目的是为了其中模式标志集的类型进行转换，也即
// 从 termios 的长整数类型转换为 termio 的短整数类型。
        tmp_termio.c_iflag = tty->termios.c_iflag;
        tmp_termio.c_oflag = tty->termios.c_oflag;
        tmp_termio.c_cflag = tty->termios.c_cflag;
        tmp_termio.c_lflag = tty->termios.c_lflag;
// 两种结构的 c_line 和 c_cc[]字段是完全相同的。
        tmp_termio.c_line = tty->termios.c_line;
        for(i=0 ; i < NCC ; i++)
                tmp_termio.c_cc[i] = tty->termios.c_cc[i];
// 最后复制指定 tty 结构中的 termio 结构信息到用户 termio 结构缓冲区。
        for (i=0 ; i< (sizeof (*termio)) ; i++)
                put_fs_byte( ((char *)&tmp_termio)[i] , i+(char *)termio );
        return 0;
}

/*
 * This only works as the 386 is low-byt-first
 */
/*
 * 下面的 termio 设置函数仅在 386 低字节在前的方式下可用。
 */
//// 设置终端 termio 结构信息。
// 参数：tty - 指定终端的 tty 结构指针；termio - 用户数据区 termio 结构指针。
// 将用户缓冲区 termio 的信息复制到终端的 termios 结构中。返回 0 。
static int set_termio(struct tty_struct * tty, struct termio * termio)
```

```
{
        int i;
        struct termio tmp_termio;

// 首先复制用户数据区中 termio 结构信息到临时 termio 结构中。
        for (i=0 ; i< (sizeof (*termio)) ; i++)
                ((char *)&tmp_termio)[i]=get_fs_byte(i+(char *)termio);
// 再将 termio 结构的信息复制到 tty 的 termios 结构中。目的是为了其中模式标志集的类型进行转换，
// 也即从 termio 的短整数类型转换成 termios 的长整数类型。
        *(unsigned short *)&tty->termios.c_iflag = tmp_termio.c_iflag;
        *(unsigned short *)&tty->termios.c_oflag = tmp_termio.c_oflag;
        *(unsigned short *)&tty->termios.c_cflag = tmp_termio.c_cflag;
        *(unsigned short *)&tty->termios.c_lflag = tmp_termio.c_lflag;
// 两种结构的 c_line 和 c_cc[]字段是完全相同的。
        tty->termios.c_line = tmp_termio.c_line;
        for(i=0 ; i < NCC ; i++)
                tty->termios.c_cc[i] = tmp_termio.c_cc[i];
// 用户可能已修改了 tty 的串行口传输波特率，所以根据 termios 结构中的控制模式标志集 c_cflag
// 修改串行芯片 UART 的传输波特率。
        change_speed(tty);
        return 0;
}

//// tty 终端设备的 ioctl 函数。
// 参数：dev - 设备号；cmd - ioctl 命令；arg - 操作参数指针。
int tty_ioctl(int dev, int cmd, int arg)
{
        struct tty_struct * tty;
// 首先取 tty 的子设备号。如果主设备号是 5(tty 终端)，则进程的 tty 字段即是子设备号；如果进程
// 的 tty 子设备号是负数，表明该进程没有控制终端，也即不能发出该 ioctl 调用，出错死机。
        if (MAJOR(dev) == 5) {
                dev=current->tty;
                if (dev<0)
                        panic("tty_ioctl: dev<0");
// 否则直接从设备号中取出子设备号。
        } else
                dev=MINOR(dev);
// 子设备号可以是 0(控制台终端)、1(串口 1 终端)、2(串口 2 终端)。
// 让 tty 指向对应子设备号的 tty 结构。
        tty = dev + tty_table;
// 根据 tty 的 ioctl 命令进行分别处理。
        switch (cmd) {
                case TCGETS:
//取相应终端 termios 结构中的信息。
                        return get_termios(tty,(struct termios *) arg);
                case TCSETSF:
// 在设置 termios 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新(清空)输入队列。
// 再设置。
                        flush(&tty->read_q); /* fallthrough */
                case TCSETSW:
// 在设置终端 termios 的信息之前，需要先等待输出队列中所有数据处理完(耗尽)。对于修改参数
// 会影响输出的情况，就需要使用这种形式。
                        wait_until_sent(tty); /* fallthrough */
```

```
                case TCSETS:
// 设置相应终端 termios 结构中的信息。
                        return set_termios(tty,(struct termios *) arg);
                case TCGETA:
// 取相应终端 termio 结构中的信息。
                        return get_termio(tty,(struct termio *) arg);
                case TCSETAF:
// 在设置 termio 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新(清空)输入队列。
// 再设置。
                        flush(&tty->read_q); /* fallthrough */
                case TCSETAW:
// 在设置终端 termio 的信息之前，需要先等待输出队列中所有数据处理完(耗尽)。对于修改参数
// 会影响输出的情况，就需要使用这种形式。
                        wait_until_sent(tty); /* fallthrough */ /* 继续执行 */
                case TCSETA:
// 设置相应终端 termio 结构中的信息。
                        return set_termio(tty,(struct termio *) arg);
                case TCSBRK:
// 等待输出队列处理完毕(空)，如果参数值是 0，则发送一个 break。
                        if (!arg) {
                                wait_until_sent(tty);
                                send_break(tty);
                        }
                        return 0;
                case TCXONC:
// 开始/停止控制。如果参数值是 0，则挂起输出；如果是 1，则重新开启挂起的输出；如果是 2，则挂
起
// 输入；如果是 3，则重新开启挂起的输入。
                        return -EINVAL; /* not implemented */  /* 未实现 */
                case TCFLSH:
//刷新已写输出但还没发送或已收但还没有读数据。如果参数是 0，则刷新(清空)输入队列；如果是 1，
// 则刷新输出队列；如果是 2，则刷新输入和输出队列。
                        if (arg==0)
                                flush(&tty->read_q);
                        else if (arg==1)
                                flush(&tty->write_q);
                        else if (arg==2) {
                                flush(&tty->read_q);
                                flush(&tty->write_q);
                        } else
                                return -EINVAL;
                        return 0;
                case TIOCEXCL:
// 设置终端串行线路专用模式。
                        return -EINVAL; /* not implemented */  /* 未实现 */
                case TIOCNXCL:
// 复位终端串行线路专用模式。
                        return -EINVAL; /* not implemented */  /* 未实现 */
                case TIOCSCTTY:
// 设置 tty 为控制终端。(TIOCNOTTY - 禁止 tty 为控制终端)。
                        return -EINVAL; /* set controlling term NI */  /* 设置控制终端 NI */
                case TIOCGPGRP:                                   // NI - Not Implemented。
// 读取指定终端设备进程的组 id。首先验证用户缓冲区长度，然后复制 tty 的 pgrp 字段到用户缓冲区。
```

```
                        verify_area((void *) arg,4);
                        put_fs_long(tty->pgrp,(unsigned long *) arg);
                        return 0;
                case TIOCSPGRP:
// 设置指定终端设备进程的组 id。
                        tty->pgrp=get_fs_long((unsigned long *) arg);
                        return 0;
                case TIOCOUTQ:
// 返回输出队列中还未送出的字符数。首先验证用户缓冲区长度，然后复制队列中字符数给用户。
                        verify_area((void *) arg,4);
                        put_fs_long(CHARS(tty->write_q),(unsigned long *) arg);
                        return 0;
                case TIOCINQ:
// 返回输入队列中还未读取的字符数。首先验证用户缓冲区长度，然后复制队列中字符数给用户。
                        verify_area((void *) arg,4);
                        put_fs_long(CHARS(tty->secondary),
                                (unsigned long *) arg);
                        return 0;
                case TIOCSTI:
// 模拟终端输入。该命令以一个指向字符的指针作为参数，并假装该字符是在终端上键入的。用户必须
// 在该控制终端上具有超级用户权限或具有读许可权限。
                        return -EINVAL; /* not implemented */  /* 未实现 */
                case TIOCGWINSZ:
// 读取终端设备窗口大小信息（参见 termios.h 中的 winsize 结构）。
                        return -EINVAL; /* not implemented */  /* 未实现 */
                case TIOCSWINSZ:
// 设置终端设备窗口大小信息（参见 winsize 结构）。
                        return -EINVAL; /* not implemented */  /* 未实现 */
                case TIOCMGET:
// 返回 modem 状态控制引线的当前状态比特位标志集（参见 termios.h 中 185-196 行）。
                        return -EINVAL; /* not implemented */  /* 未实现 */
                case TIOCMBIS:
// 设置单个 modem 状态控制引线的状态(true 或 false)。
                        return -EINVAL; /* not implemented */  /* 未实现 */
                case TIOCMBIC:
// 复位单个 modem 状态控制引线的状态。
                        return -EINVAL; /* not implemented */  /* 未实现 */
                case TIOCMSET:
// 设置 modem 状态引线的状态。如果某一比特位置位，则 modem 对应的状态引线将置为有效。
                        return -EINVAL; /* not implemented */  /* 未实现 */
                case TIOCGSOFTCAR:
// 读取软件载波检测标志(1 - 开启；0 - 关闭)。
                        return -EINVAL; /* not implemented */  /* 未实现 */
                case TIOCSSOFTCAR:
// 设置软件载波检测标志(1 - 开启；0 - 关闭)。
                        return -EINVAL; /* not implemented */  /* 未实现 */
                default:
                        return -EINVAL;
        }
}
```

7.6.3 其它信息

7.6.3.1 波特率与波特率因子

波特率 = 1.8432MHz /(16 * 波特率因子)。程序中波特率与波特率因子的对应关系见下表所示。

表 7.3 波特率与波特率因子对应表

波特率	波特率因子		波特率	波特率因子	
	MSB,LSB	合并值		MSB,LSB	合并值
50	0x09,0x00	2304	1200	0x00,0x60	96
75	0x06,0x00	1536	1800	0x00,0x40	64
110	0x04,0x17	1047	2400	0x00,0x30	48
134.5	0x03,0x59	857	4800	0x00,0x18	24
150	0x03,0x00	768	9600	0x00,0x1c	12
200	0x02,0x40	576	19200	0x00,0x06	6
300	0x01,0x80	384	38400	0x00,0x03	3
600	0x00,0xc0	192			

7.7 keyboard.S 文件

7.7.1 功能描述

该键盘驱动汇编程序主要包括键盘中断处理程序。在英文惯用法中，make 表示键被按下；break 表示键被松开(放开)。

对于 AT 键盘的扫描码，当键按下时，则对应键的扫描码被送出，但当键松开时，将会发送两个字节，第一个是 0xf0，第 2 个还是按下时的扫描码。为了向下的兼容性，设计人员将 AT 键盘发出的扫描码转换成了老式 PC/XT 标准键盘的扫描码。因此这里仅对 PC/XT 的扫描码进行处理即可。

7.7.2 代码注释

列表 7.7 linux/kernel/chr_drv/keyboard.S 文件

```
/*
 *  linux/kernel/keyboard.S
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      Thanks to Alfred Leung for US keyboard patches
 *              Wolfgang Thiel for German keyboard patches
 *              Marc Corsini for the French keyboard
 */
/*
 * 感谢 Alfred Leung 添加了 US 键盘补丁程序；
```

```
 *     Wolfgang Thiel 添加了德语键盘补丁程序；
 *     Marc Corsini 添加了法文键盘补丁程序。
 */

#include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）可选项。

.text
.globl _keyboard_interrupt

/*
 * these are for the keyboard read functions
 */
/*
 * 以下这些是用于键盘读操作。
 */
// size 是键盘缓冲区的长度（字节数）。
size    = 1024          /* must be a power of two ! And MUST be the same
                           as in tty_io.c !!!! */
                        /* 数值必须是 2 的次方！并且与 tty_io.c 中的值匹配!!!! */
// 以下这些是缓冲队列结构中的偏移量 */
head = 4                // 缓冲区中头指针字段偏移。
tail = 8                // 缓冲区中尾指针字段偏移。
proc_list = 12          // 等待该缓冲队列的进程字段偏移。
buf = 16                // 缓冲区字段偏移。

// mode 是键盘特殊键的按下状态标志。
// 表示大小写转换键(caps)、交换键(alt)、控制键(ctrl)和换档键(shift)的状态。
// 位 7 caps 键按下；
// 位 6 caps 键的状态(应该与 leds 中的对应标志位一样)；
// 位 5 右 alt 键按下；
// 位 4 左 alt 键按下；
// 位 3 右 ctrl 键按下；
// 位 2 左 ctrl 键按下；
// 位 1 右 shift 键按下；
// 位 0 左 shift 键按下。
mode:   .byte 0         /* caps, alt, ctrl and shift mode */
// 数字锁定键(num-lock)、大小写转换键(caps-lock)和滚动锁定键(scroll-lock)的 LED 发光管状态。
// 位 7-3 全 0 不用；
// 位 2 caps-lock；
// 位 1 num-lock(初始置 1，也即设置数字锁定键(num-lock)发光管为亮)；
// 位 0 scroll-lock。
leds:   .byte 2         /* num-lock, caps, scroll-lock mode (nom-lock on) */
// 当扫描码是 0xe0 或 0xe1 时，置该标志。表示其后还跟随着 1 个或 2 个字符扫描码，参见列表后说明。
// 位 1 =1 收到 0xe1 标志；
// 位 0 =1 收到 0xe0 标志。
e0:     .byte 0

/*
 *  con_int is the real interrupt routine that reads the
 *  keyboard scan-code and converts it into the appropriate
 *  ascii character(s).
 */
/*
```

```
 * con_int 是实际的中断处理子程序，用于读键盘扫描码并将其转换
 * 成相应的 ascii 字符。
 */
//// 键盘中断处理程序入口点。
37 _keyboard_interrupt:
38         pushl %eax
39         pushl %ebx
40         pushl %ecx
41         pushl %edx
42         push %ds
43         push %es
44         movl $0x10,%eax           // 将 ds、es 段寄存器置为内核数据段。
45         mov %ax,%ds
46         mov %ax,%es
47         xorl %al,%al             /* %eax is scan code */  /* eax 中是扫描码 */
48         inb $0x60,%al            // 读取扫描码➔al。
49         cmpb $0xe0,%al           // 该扫描码是 0xe0 吗？如果是则跳转到设置 e0 标志代码处。
50         je set_e0
51         cmpb $0xe1,%al           // 扫描码是 0xe1 吗？如果是则跳转到设置 e1 标志代码处。
52         je set_e1
53         call key_table(,%eax,4)  // 调用键处理程序 ker_table + eax * 4（参见下面 502 行）。
54         movb $0,e0               // 复位 e0 标志。
// 下面这段代码(55-65 行)是针对使用 8255A 的 PC 标准键盘电路进行硬件复位处理。端口 0x61 是
// 8255A 输出口 B 的地址，该输出端口的第 7 位(PB7)用于禁止和允许对键盘数据的处理。
// 这段程序用于对收到的扫描码做出应答。方法是首先禁止键盘，然后立刻重新允许键盘工作。
55 e0_e1:  inb $0x61,%al            // 取 PPI 端口 B 状态，其位 7 用于允许/禁止(0/1)键盘。
56         jmp 1f                   // 延迟一会。
57 1:      jmp 1f
58 1:      orb $0x80,%al            // al 位 7 置位(禁止键盘工作)。
59         jmp 1f                   // 再延迟一会。
60 1:      jmp 1f
61 1:      outb %al,$0x61           // 使 PPI PB7 位置位。
62         jmp 1f                   // 延迟一会。
63 1:      jmp 1f
64 1:      andb $0x7F,%al           // al 位 7 复位。
65         outb %al,$0x61           // 使 PPI PB7 位复位（允许键盘工作）。

66         movb $0x20,%al           // 向 8259 中断芯片发送 EOI(中断结束)信号。
67         outb %al,$0x20

68         pushl $0                 // 控制台 tty 号=0，作为参数入栈。
69         call _do_tty_interrupt   // 将收到的数据复制成规范模式数据并存放在规范字符缓冲队列
中。
70         addl $4,%esp             // 丢弃入栈的参数，弹出保留的寄存器，并中断返回。
71         pop %es
72         pop %ds
73         popl %edx
74         popl %ecx
75         popl %ebx
76         popl %eax
77         iret
78 set_e0: movb $1,e0               // 收到扫描前导码 0xe0 时，设置 e0 标志（位 0）。
79         jmp e0_e1
```

```
set_e1: movb $2,e0              // 收到扫描前导码 0xe1 时，设置 e1 标志（位 1）。
        jmp e0_e1

/*
 * This routine fills the buffer with max 8 bytes, taken from
 * %ebx:%eax. (%edx is high). The bytes are written in the
 * order %al,%ah,%eal,%eah,%bl,%bh ... until %eax is zero.
 */
/*
 * 下面该子程序把 ebx:eax 中的最多 8 个字符添入缓冲队列中。(edx 是
 * 所写入字符的顺序是 al,ah,eal,eah,bl,bh...直到 eax 等于 0。
 */
put_queue:
        pushl %ecx              // 保存 ecx，edx 内容。
        pushl %edx              // 取控制台 tty 结构中读缓冲队列指针。
        movl _table_list,%edx   # read-queue for console
        movl head(%edx),%ecx    // 取缓冲队列中头指针➔ecx。
1:      movb %al,buf(%edx,%ecx) // 将 al 中的字符放入缓冲队列头指针位置处。
        incl %ecx               // 头指针前移 1 字节。
        andl $size-1,%ecx       // 以缓冲区大小调整头指针(若超出则返回缓冲区开始)。
        cmpl tail(%edx),%ecx    # buffer full - discard everything
                                // 头指针==尾指针吗(缓冲队列满)？
        je 3f                   // 如果已满，则后面未放入的字符全抛弃。
        shrdl $8,%ebx,%eax      // 将 ebx 中 8 位比特位右移 8 位到 eax 中，但 ebx 不变。
        je 2f                   // 还有字符吗？若没有(等于 0)则跳转。
        shrl $8,%ebx            // 将 ebx 中比特位右移 8 位，并跳转到标号 1 继续操作。
        jmp 1b
2:      movl %ecx,head(%edx)    // 若已将所有字符都放入了队列，则保存头指针。
        movl proc_list(%edx),%ecx // 该队列的等待进程指针？
        testl %ecx,%ecx         // 检测任务结构指针是否为空(有等待该队列的进程
吗？)。
        je 3f                   // 无，则跳转；
        movl $0,(%ecx)          // 有，则置该进程为可运行就绪状态(唤醒该进程)。
3:      popl %edx               // 弹出保留的寄存器并返回。
        popl %ecx
        ret

// 下面这段代码根据 ctrl 或 alt 的扫描码，分别设置模式标志中相应位。如果该扫描码之前收到过
// 0xe0 扫描码(e0 标志置位)，则说明按下的是键盘右边的 ctrl 或 alt 键，则对应设置 ctrl 或 alt
// 在模式标志 mode 中的比特位。
ctrl:   movb $0x04,%al          // 0x4 是模式标志 mode 中左 ctrl 键对应的比特位(位 2)。
        jmp 1f
alt:    movb $0x10,%al          // 0x10 是模式标志 mode 中左 alt 键对应的比特位(位 4)。
1:      cmpb $0,e0              // e0 标志置位了吗(按下的是右边的 ctrl 或 alt 键吗)？
        je 2f                   // 不是则转。
        addb %al,%al            // 是，则改成置相应右键的标志位(位 3 或位 5)。
2:      orb %al,mode            // 设置模式标志 mode 中对应的比特位。
        ret
// 这段代码处理 ctrl 或 alt 键松开的扫描码，对应复位模式标志 mode 中的比特位。在处理时要根据
// e0 标志是否置位来判断是否是键盘右边的 ctrl 或 alt 键。
unctrl: movb $0x04,%al          // 模式标志 mode 中左 ctrl 键对应的比特位(位 2)。
        jmp 1f
unalt:  movb $0x10,%al          // 0x10 是模式标志 mode 中左 alt 键对应的比特位(位 4)。
```

```
1:      cmpb $0,e0                  // e0 标志置位了吗(释放的是右边的 ctrl 或 alt 键吗)?
        je 2f                       // 不是，则转。
        addb %al,%al                // 是，则该成复位相应右键的标志位(位 3 或位 5)。
2:      notb %al                    // 复位模式标志 mode 中对应的比特位。
        andb %al,mode
        ret

lshift:
        orb $0x01,mode              // 是左 shift 键按下，设置 mode 中对应的标志位(位 0)。
        ret
unlshift:
        andb $0xfe,mode             // 是左 shift 键松开，复位 mode 中对应的标志位(位 0)。
        ret
rshift:
        orb $0x02,mode              // 是右 shift 键按下，设置 mode 中对应的标志位(位 1)。
        ret
unrshift:
        andb $0xfd,mode             // 是右 shift 键松开，复位 mode 中对应的标志位(位 1)。
        ret

caps:   testb $0x80,mode            // 测试模式标志 mode 中位 7 是否已经置位(按下状态)。
        jne 1f                      // 如果已处于按下状态，则返回(ret)。
        xorb $4,leds                // 翻转 leds 标志中 caps-lock 比特位(位 2)。
        xorb $0x40,mode             // 翻转 mode 标志中 caps 键按下的比特位(位 6)。
        orb $0x80,mode              // 设置 mode 标志中 caps 键已按下标志位(位 7)。
// 这段代码根据 leds 标志，开启或关闭 LED 指示器。
set_leds:
        call kb_wait                // 等待键盘控制器输入缓冲空。
        movb $0xed,%al      /* set leds command */  /* 设置 LED 的命令 */
        outb %al,$0x60              // 发送键盘命令 0xed 到 0x60 端口。
        call kb_wait                // 等待键盘控制器输入缓冲空。
        movb leds,%al               // 取 leds 标志，作为参数。
        outb %al,$0x60              // 发送该参数。
        ret
uncaps: andb $0x7f,mode             // caps 键松开，则复位模式标志 mode 中的对应位(位 7)。
        ret
scroll:
        xorb $1,leds                // scroll 键按下，则翻转 leds 标志中的对应位(位 0)。
        jmp set_leds                // 根据 leds 标志重新开启或关闭 LED 指示器。
num:    xorb $2,leds                // num 键按下，则翻转 leds 标志中的对应位(位 1)。
        jmp set_leds                // 根据 leds 标志重新开启或关闭 LED 指示器。

/*
 *  curosr-key/numeric keypad cursor keys are handled here.
 *  checking for numeric keypad etc.
 */
/*
 * 这里处理方向键/数字小键盘方向键，检测数字小键盘等。
 */
cursor:
        subb $0x47,%al      // 扫描码是小数字键盘上的键(其扫描码>=0x47)发出的?
        jb 1f               // 如果小于则不处理，返回。
        cmpb $12,%al        // 如果扫描码 > 0x53(0x53 - 0x47= 12)，则
```

```
171         ja 1f                   // 扫描码值超过 83(0x53)，不处理，返回。
172         jne cur2                /* check for ctrl-alt-del */ /* 检查是否 ctrl-alt-del */
                                    // 如果等于 12，则说明 del 键已被按下，则继续判断 ctrl
                                    // 和 alt 是否也同时按下。
173         testb $0x0c,mode        // 有 ctrl 键按下吗？
174         je cur2                 // 无，则跳转。
175         testb $0x30,mode        // 有 alt 键按下吗？
176         jne reboot              // 有，则跳转到重启动处理。
177 cur2:   cmpb $0x01,e0           /* e0 forces cursor movement */ /* e0 置位表示光标移动 */
                                    // e0 标志置位了吗？
178         je cur                  // 置位了，则跳转光标移动处理处 cur。
179         testb $0x02,leds        /* not num-lock forces cursor */ /* num-lock 键则不许 */
                                    // 测试 leds 中标志 num-lock 键标志是否置位。
180         je cur                  // 如果没有置位(num 的 LED 不亮)，则也进行光标移动处理。
181         testb $0x03,mode        /* shift forces cursor */ /* shift 键也使光标移动 */
                                    // 测试模式标志 mode 中 shift 按下标志。
182         jne cur                 // 如果有 shift 键按下，则也进行光标移动处理。
183         xorl %ebx,%ebx          // 否则查询扫数字表(199 行)，取对应键的数字 ASCII 码。
184         movb num_table(%eax),%al    // 以 eax 作为索引值，取对应数字字符➔al。
185         jmp put_queue           // 将该字符放入缓冲队列中。
186 1:      ret
187
    // 这段代码处理光标的移动。
188 cur:    movb cur_table(%eax),%al  // 取光标字符表中相应键的代表字符➔al。
189         cmpb $'9,%al              // 若该字符<='9'，说明是上一页、下一页、插入或删除键，
190         ja ok_cur                 // 则功能字符序列中要添入字符'~'。
191         movb $'~,%ah
192 ok_cur: shll $16,%eax             // 将 ax 中内容移到 eax 高字中。
193         movw $0x5b1b,%ax          // 在 ax 中放入'esc ['字符，与 eax 高字中字符组成移动序列。
194         xorl %ebx,%ebx
195         jmp put_queue             // 将该字符放入缓冲队列中。
196
197 #if defined(KBD_FR)
198 num_table:
199         .ascii "789 456 1230."    // 数字小键盘上键对应的数字 ASCII 码表。
200 #else
201 num_table:
202         .ascii "789 456 1230,"
203 #endif
204 cur_table:
205         .ascii "HA5 DGC YB623"    // 数字小键盘上方向键或插入删除键对应的移动表示字符表。
206
207 /*
208  * this routine handles function keys
209  */
    // 下面子程序处理功能键。
210 func:
211         pushl %eax
212         pushl %ecx
213         pushl %edx
214         call _show_stat         // 调用显示各任务状态函数(kernl/sched.c, 37)。
215         popl %edx
216         popl %ecx
```

```
        popl %eax
        subb $0x3B,%al          // 功能键'F1'的扫描码是 0x3B，因此此时 al 中是功能键索引号。
        jb end_func             // 如果扫描码小于 0x3b，则不处理，返回。
        cmpb $9,%al             // 功能键是 F1-F10？
        jbe ok_func             // 是，则跳转。
        subb $18,%al            // 是功能键 F11，F12 吗？
        cmpb $10,%al            // 是功能键 F11？
        jb end_func             // 不是，则不处理，返回。
        cmpb $11,%al            // 是功能键 F12？
        ja end_func             // 不是，则不处理，返回。
ok_func:
        cmpl $4,%ecx            /* check that there is enough room */ * 检查是否有足够空间
*/
        jl end_func             // 需要放入 4 个字符序列，如果放不下，则返回。
        movl func_table(,%eax,4),%eax  // 取功能键对应字符序列。
        xorl %ebx,%ebx
        jmp put_queue           // 放入缓冲队列中。
end_func:
        ret

/*
 * function keys send F1:'esc [ [ A' F2:'esc [ [ B' etc.
 */
/*
 * 功能键发送的扫描码，F1 键为：'esc [ [ A'， F2 键为：'esc [ [ B'等。
 */
func_table:
        .long 0x415b5b1b,0x425b5b1b,0x435b5b1b,0x445b5b1b
        .long 0x455b5b1b,0x465b5b1b,0x475b5b1b,0x485b5b1b
        .long 0x495b5b1b,0x4a5b5b1b,0x4b5b5b1b,0x4c5b5b1b

// 扫描码-ASCII 字符映射表。
// 根据在 config.h 中定义的键盘类型(FINNISH，US，GERMEN，FRANCH)，将相应键的扫描码映射
// 到 ASCII 字符。
#if     defined(KBD_FINNISH)
// 以下是芬兰语键盘的扫描码映射表。
key_map:
        .byte 0,27              // 扫描码 0x00,0x01 对应的 ASCII 码；
        .ascii "1234567890+'"   // 扫描码 0x02,...0x0c,0x0d 对应的 ASCII 码，以下类似。
        .byte 127,9
        .ascii "qwertyuiop}"
        .byte 0,13,0
        .ascii "asdfghjkl|{"
        .byte 0,0
        .ascii "'zxcvbnm,.-"
        .byte 0,'*,0,32         /* 36-39 */   /* 扫描码 0x36-0x39 对应的 ASCII 码 */
        .fill 16,1,0            /* 3A-49 */   /* 扫描码 0x3A-0x49 对应的 ASCII 码 */
        .byte '-,0,0,0,'+       /* 4A-4E */   /* 扫描码 0x4A-0x4E 对应的 ASCII 码 */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */   /* 扫描码 0x4F-0x55 对应的 ASCII 码 */
        .byte '<
        .fill 10,1,0

// shift 键同时按下时的映射表。
```

```
shift_map:
        .byte 0,27
        .ascii "!\"#$%&/()=?`"
        .byte 127,9
        .ascii "QWERTYUIOP]^"
        .byte 13,0
        .ascii "ASDFGHJKL\\["
        .byte 0,0
        .ascii "*ZXCVBNM;:_"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '>
        .fill 10,1,0

// alt 键同时按下时的映射表。
alt_map:
        .byte 0,0
        .ascii "\0@\0$\0\0{[]}\\\0"
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte '~,13,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0,0,0           /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte 0,0,0,0,0         /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '|
        .fill 10,1,0

#elif defined(KBD_US)

// 以下是美式键盘的扫描码映射表。
key_map:
        .byte 0,27
        .ascii "1234567890-="
        .byte 127,9
        .ascii "qwertyuiop[]"
        .byte 13,0
        .ascii "asdfghjkl;'"
        .byte '`,0
        .ascii "\\zxcvbnm,./"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '<
        .fill 10,1,0
```

```
shift_map:
        .byte 0,27
        .ascii "!@#$%^&*()_+"
        .byte 127,9
        .ascii "QWERTYUIOP{}"
        .byte 13,0
        .ascii "ASDFGHJKL:\""
        .byte '~,0
        .ascii "|ZXCVBNM<>?"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '>
        .fill 10,1,0

alt_map:
        .byte 0,0
        .ascii "\0@\0$\0\0{[]}\\\0"
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte '~,13,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0,0,0           /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte 0,0,0,0,0         /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '|
        .fill 10,1,0

#elif defined(KBD_GR)

// 以下是德语键盘的扫描码映射表。
key_map:
        .byte 0,27
        .ascii "1234567890\\'"
        .byte 127,9
        .ascii "qwertzuiop@+"
        .byte 13,0
        .ascii "asdfghjkl[]^"
        .byte 0,'#
        .ascii "yxcvbnm,.-"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '<
        .fill 10,1,0


shift_map:
```

```
        .byte 0,27
        .ascii "!\"#$%&/()=?`"
        .byte 127,9
        .ascii "QWERTZUIOP\\*"
        .byte 13,0
        .ascii "ASDFGHJKL{}~"
        .byte 0,''
        .ascii "YXCVBNM;:_"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '>
        .fill 10,1,0

alt_map:
        .byte 0,0
        .ascii "\0@\0$\0\0{[]}\\\0"
        .byte 0,0
        .byte '@,0,0,0,0,0,0,0,0,0,0,0
        .byte '~,13,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0
        .byte 0,0,0,0,0,0,0,0,0,0,0
        .byte 0,0,0,0           /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte 0,0,0,0,0         /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '|
        .fill 10,1,0


#elif defined(KBD_FR)

// 以下是法语键盘的扫描码映射表。
key_map:
        .byte 0,27
        .ascii "&{\"'(-}_/@)="
        .byte 127,9
        .ascii "azertyuiop^$"
        .byte 13,0
        .ascii "qsdfghjklm|"
        .byte '`,0,42           /* coin sup gauche, don't know, [*|mu] */
        .ascii "wxcvbn,;:!"
        .byte 0,'*,0,32         /* 36-39 */
        .fill 16,1,0            /* 3A-49 */
        .byte '-,0,0,0,'+       /* 4A-4E */
        .byte 0,0,0,0,0,0,0     /* 4F-55 */
        .byte '<
        .fill 10,1,0

shift_map:
        .byte 0,27
```

```
416         .ascii "1234567890]+"
417         .byte 127,9
418         .ascii "AZERTYUIOP<>"
419         .byte 13,0
420         .ascii "QSDFGHJKLM%"
421         .byte '~,0,'#
422         .ascii "WXCVBN?./\\"
423         .byte 0,'*,0,32         /* 36-39 */
424         .fill 16,1,0            /* 3A-49 */
425         .byte '-,0,0,0,'+       /* 4A-4E */
426         .byte 0,0,0,0,0,0,0     /* 4F-55 */
427         .byte '>
428         .fill 10,1,0
429
430 alt_map:
431         .byte 0,0
432         .ascii "\0~#{[|`\\^@]}"
433         .byte 0,0
434         .byte '@,0,0,0,0,0,0,0,0,0,0
435         .byte '~,13,0
436         .byte 0,0,0,0,0,0,0,0,0,0,0
437         .byte 0,0
438         .byte 0,0,0,0,0,0,0,0,0,0,0
439         .byte 0,0,0,0           /* 36-39 */
440         .fill 16,1,0            /* 3A-49 */
441         .byte 0,0,0,0,0         /* 4A-4E */
442         .byte 0,0,0,0,0,0,0     /* 4F-55 */
443         .byte '|
444         .fill 10,1,0
445
446 #else
447 #error "KBD-type not defined"
448 #endif
449 /*
450  * do_self handles "normal" keys, ie keys that don't change meaning
451  * and which have just one character returns.
452  */
    /*
     * do_self 用于处理“普通”键，也即含义没有变化并且只有一个字符返回的键。
     */
453 do_self:
    // 454-460 行用于根据模式标志 mode 选择 alt_map、shift_map 或 key_map 映射表之一。
454         lea alt_map,%ebx                // alt 键同时按下时的映射表基址 alt_map➔ebx。
455         testb $0x20,mode                /* alt-gr */  /* 右 alt 键同时按下了? */
456         jne 1f                          // 是，则向前跳转到标号 1 处。
457         lea shift_map,%ebx              // shift 键同时按下时的映射表基址 shift_map➔ebx。
458         testb $0x03,mode                // 有 shift 键同时按下了吗?
459         jne 1f                          // 有，则向前跳转到标号 1 处。
460         lea key_map,%ebx                // 否则使用普通映射表 key_map。
    // 取映射表中对应扫描码的 ASCII 字符，若没有对应字符，则返回(转 none)。
461 1:      movb (%ebx,%eax),%al            // 将扫描码作为索引值，取对应的 ASCII 码➔al。
462         orb %al,%al                     // 检测看是否有对应的 ASCII 码。
463         je none                         // 若没有(对应的 ASCII 码=0)，则返回。
```

```
// 若 ctrl 键已按下或 caps 键锁定，并且字符在'a'-'}'(0x61-0x7D)范围内，则将其转成大写字符
// (0x41-0x5D)。
        testb $0x4c,mode            /* ctrl or caps */ /* 控制键已按下或 caps 亮？*/
        je 2f                       // 没有，则向前跳转标号 2 处。
        cmpb $'a,%al                // 将 al 中的字符与'a'比较。
        jb 2f                       // 若 al 值<'a'，则转标号 2 处。
        cmpb $'},%al                // 将 al 中的字符与'}'比较。
        ja 2f                       // 若 al 值>'}'，则转标号 2 处。
        subb $32,%al                // 将 al 转换为大写字符(减 0x20)。
// 若 ctrl 键已按下，并且字符在'`'--'_'(0x40-0x5F)之间(是大写字符)，则将其转换为控制字符
// (0x00-0x1F)。
2:      testb $0x0c,mode            /* ctrl */   /* ctrl 键同时按下了吗？*/
        je 3f                       // 若没有则转标号 3。
        cmpb $64,%al                // 将 al 与'@'(64)字符比较(即判断字符所属范围)。
        jb 3f                       // 若值<'@'，则转标号 3。
        cmpb $64+32,%al             // 将 al 与'`'(96)字符比较(即判断字符所属范围)。
        jae 3f                      // 若值>='`'，则转标号 3。
        subb $64,%al                // 否则 al 值减 0x40，
                                    // 即将字符转换为 0x00-0x1f 之间的控制字符。
// 若左 alt 键同时按下，则将字符的位 7 置位。
3:      testb $0x10,mode            /* left alt */   /* 左 alt 键同时按下？*/
        je 4f                       // 没有，则转标号 4。
        orb $0x80,%al               // 字符的位 7 置位。
// 将 al 中的字符放入读缓冲队列中。
4:      andl $0xff,%eax             // 清 eax 的高字和 ah。
        xorl %ebx,%ebx              // 清 ebx。
        call put_queue              // 将字符放入缓冲队列中。
none:   ret

/*
 * minus has a routine of it's own, as a 'E0h' before
 * the scan code for minus means that the numeric keypad
 * slash was pushed.
 */
/*
 * 减号有它自己的处理子程序，因为在减号扫描码之前的 0xe0
 * 意味着按下了数字小键盘上的斜杠键。
 */
minus:  cmpb $1,e0                  // e0 标志置位了吗？
        jne do_self                 // 没有，则调用 do_self 对减号符进行普通处理。
        movl $'/,%eax               // 否则用'/'替换减号'-'➔al。
        xorl %ebx,%ebx
        jmp put_queue               // 并将字符放入缓冲队列中。

/*
 * This table decides which routine to call when a scan-code has been
 * gotten. Most routines just call do_self, or none, depending if
 * they are make or break.
 */
/* 下面是一张子程序地址跳转表。当取得扫描码后就根据此表调用相应的扫描码处理子程序。
 * 大多数调用的子程序是 do_self，或者是 none，这起决于是按键(make)还是释放键(break)。
 */
key_table:
```

```
        .long none,do_self,do_self,do_self      /* 00-03 s0 esc 1 2 */
        .long do_self,do_self,do_self,do_self   /* 04-07 3 4 5 6 */
        .long do_self,do_self,do_self,do_self   /* 08-0B 7 8 9 0 */
        .long do_self,do_self,do_self,do_self   /* 0C-0F + ' bs tab */
        .long do_self,do_self,do_self,do_self   /* 10-13 q w e r */
        .long do_self,do_self,do_self,do_self   /* 14-17 t y u i */
        .long do_self,do_self,do_self,do_self   /* 18-1B o p } ^ */
        .long do_self,ctrl,do_self,do_self      /* 1C-1F enter ctrl a s */
        .long do_self,do_self,do_self,do_self   /* 20-23 d f g h */
        .long do_self,do_self,do_self,do_self   /* 24-27 j k l | */
        .long do_self,do_self,lshift,do_self    /* 28-2B { para lshift , */
        .long do_self,do_self,do_self,do_self   /* 2C-2F z x c v */
        .long do_self,do_self,do_self,do_self   /* 30-33 b n m , */
        .long do_self,minus,rshift,do_self      /* 34-37 . - rshift * */
        .long alt,do_self,caps,func             /* 38-3B alt sp caps f1 */
        .long func,func,func,func               /* 3C-3F f2 f3 f4 f5 */
        .long func,func,func,func               /* 40-43 f6 f7 f8 f9 */
        .long func,num,scroll,cursor            /* 44-47 f10 num scr home */
        .long cursor,cursor,do_self,cursor      /* 48-4B up pgup - left */
        .long cursor,cursor,do_self,cursor      /* 4C-4F n5 right + end */
        .long cursor,cursor,cursor,cursor       /* 50-53 dn pgdn ins del */
        .long none,none,do_self,func            /* 54-57 sysreq ? < f11 */
        .long func,none,none,none               /* 58-5B f12 ? ? ? */
        .long none,none,none,none               /* 5C-5F ? ? ? ? */
        .long none,none,none,none               /* 60-63 ? ? ? ? */
        .long none,none,none,none               /* 64-67 ? ? ? ? */
        .long none,none,none,none               /* 68-6B ? ? ? ? */
        .long none,none,none,none               /* 6C-6F ? ? ? ? */
        .long none,none,none,none               /* 70-73 ? ? ? ? */
        .long none,none,none,none               /* 74-77 ? ? ? ? */
        .long none,none,none,none               /* 78-7B ? ? ? ? */
        .long none,none,none,none               /* 7C-7F ? ? ? ? */
        .long none,none,none,none               /* 80-83 ? br br br */
        .long none,none,none,none               /* 84-87 br br br br */
        .long none,none,none,none               /* 88-8B br br br br */
        .long none,none,none,none               /* 8C-8F br br br br */
        .long none,none,none,none               /* 90-93 br br br br */
        .long none,none,none,none               /* 94-97 br br br br */
        .long none,none,none,none               /* 98-9B br br br br */
        .long none,unctrl,none,none             /* 9C-9F br unctrl br br */
        .long none,none,none,none               /* A0-A3 br br br br */
        .long none,none,none,none               /* A4-A7 br br br br */
        .long none,none,unlshift,none           /* A8-AB br br unlshift br */
        .long none,none,none,none               /* AC-AF br br br br */
        .long none,none,none,none               /* B0-B3 br br br br */
        .long none,none,unrshift,none           /* B4-B7 br br unrshift br */
        .long unalt,none,uncaps,none            /* B8-BB unalt br uncaps br */
        .long none,none,none,none               /* BC-BF br br br br */
        .long none,none,none,none               /* C0-C3 br br br br */
        .long none,none,none,none               /* C4-C7 br br br br */
        .long none,none,none,none               /* C8-CB br br br br */
        .long none,none,none,none               /* CC-CF br br br br */
        .long none,none,none,none               /* D0-D3 br br br br */
```

```
        .long none,none,none,none               /* D4-D7 br br br br */
        .long none,none,none,none               /* D8-DB br ? ? ? */
        .long none,none,none,none               /* DC-DF ? ? ? ? */
        .long none,none,none,none               /* E0-E3 e0 e1 ? ? */
        .long none,none,none,none               /* E4-E7 ? ? ? ? */
        .long none,none,none,none               /* E8-EB ? ? ? ? */
        .long none,none,none,none               /* EC-EF ? ? ? ? */
        .long none,none,none,none               /* F0-F3 ? ? ? ? */
        .long none,none,none,none               /* F4-F7 ? ? ? ? */
        .long none,none,none,none               /* F8-FB ? ? ? ? */
        .long none,none,none,none               /* FC-FF ? ? ? ? */

/*
 * kb_wait waits for the keyboard controller buffer to empty.
 * there is no timeout - if the buffer doesn't empty, we hang.
 */
/*
 * 子程序 kb_wait 用于等待键盘控制器缓冲空。不存在超时处理 - 如果
 * 缓冲永远不空的话，程序就会永远等待(死掉)。
 */
kb_wait:
        pushl %eax
1:      inb $0x64,%al               // 读键盘控制器状态。
        testb $0x02,%al             // 测试输入缓冲器是否为空(等于 0)。
        jne 1b                      // 若不空，则跳转循环等待。
        popl %eax
        ret
/*
 * This routine reboots the machine by asking the keyboard
 * controller to pulse the reset-line low.
 */
/*
 * 该子程序通过设置键盘控制器，向复位线输出负脉冲，使系统复位重启(reboot)。
 */
reboot:
        call kb_wait              // 首先等待键盘控制器输入缓冲器空。
        movw $0x1234,0x472        /* don't do memory check */
        movb $0xfc,%al            /* pulse reset and A20 low */
        outb %al,$0x64            // 向系统复位和 A20 线输出负脉冲。
die:    jmp die                   // 死机。
```

7.7.3 其它信息

7.7.3.1 AT 键盘接口编程

主机系统板上所采用的键盘控制器是 intel 8042 芯片或其兼容芯片，其逻辑示意图，见下图所示。其中输出端口 P2 分别用于其它目的。位 0(P20 引脚)用于实现 CPU 的复位操作，位 1（P21 引脚）用于控制 A20 信号线的开启与否。当该输出端口位 0 为 1 时就开启（选通）了 A20 信号线，为 0 则禁止 A20 信号线。参见引导启动程序一章中对 A20 信号线的详细说明。

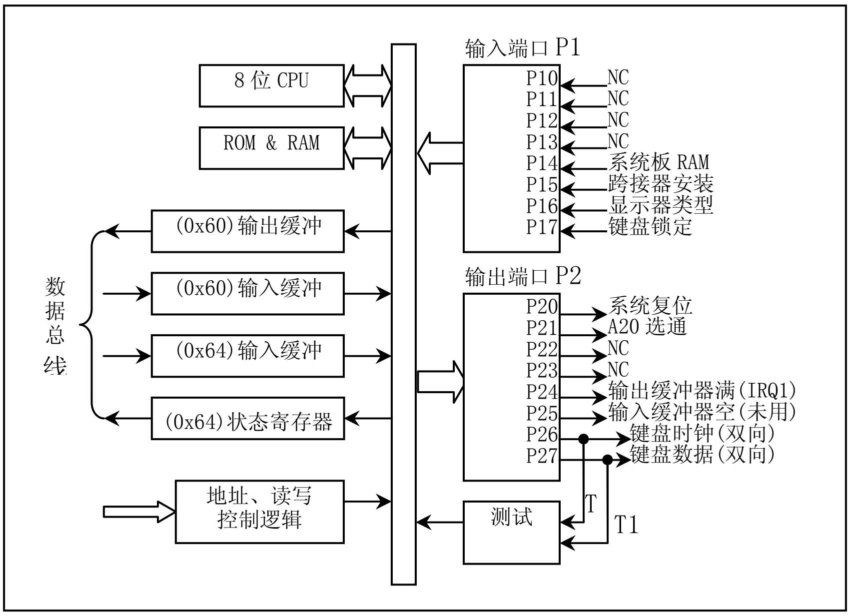


图 7.2 键盘控制器 804X 逻辑示意图

分配给键盘控制器的 IO 端口范围是 0x60-0x6f，但实际上 IBM CP/AT 使用的只有 0x60 和 0x64 两个口地址(0x61、0x62 和 0x63 用于与 XT 兼容目的)见下表所示，加上对端口的读和写操作含义不同，因此主要可有 4 种不同操作。对键盘控制器进行编程，将涉及芯片中的状态寄存器、输入缓冲器和输出缓冲器。

表 7.4 键盘控制器 804X 端口

端口	读/写	名称	用途
0x60	读	数据端口或输出缓冲器	是一个 8 位只读寄存器。当键盘控制器收到来自键盘的扫描码或命令响应时，一方面置状态寄存器位 0 = 1，另一方面产生中断 IRQ1。通常应该仅在状态端口位 0 = 1 时才读。
0x60	写	输入缓冲器	用于向键盘发送命令与/或随后的参数，或向键盘控制器写参数。键盘命令共有 10 多条，见表格后说明。通常都应该仅在状态端口位 1=0 时才写。
0x61	读/写		该端口 0x61 是 8255A 输出口 B 的地址，是针对使用/兼容 8255A 的 PC 标准键盘电路进行硬件复位处理。该端口用于对收到的扫描码做出应答。方法是首先禁止键盘，然后立刻重新允许键盘。所操作的数据为： 位 7=1 禁止键盘；=0 允许键盘； 位 6=0 迫使键盘时钟为低位，因此键盘不能发送任何数据。 位 5-0 这些位与键盘无关，是用于可编程并行接口(PPI)。
0x64	读	状态寄存器	是一个 8 位只读寄存器，其位字段含义分别为： 位 7=1 来自键盘传输数据奇偶校验错； 位 6=1 接收超时(键盘传送未产生 IRQ1)； 位 5=1 发送超时(键盘无响应)； 位 4=1 键盘接口被键盘锁禁止；[??是=0 时] 位 3=1 写入输入缓冲器中的数据是命令(通过端口 0x64)；

			=0 写入输入缓冲器中的数据是参数(通过端口 0x60); 位 2 系统标志状态：0 = 上电启动或复位；1 = 自检通过； 位 1=1 输入缓冲器满(0x60/64 口有给 8042 的数据)； 位 0=1 输出缓冲器满(数据端口 0x60 有给系统的数据)。
0x64	写	输入缓冲器	向键盘控制器写命令。可带一参数，参数从端口 0x60 写入。键盘控制器命令有 12 条，见表格后说明。

7.7.3.2 键盘命令

系统在向端口 0x60 写入 1 字节，便是发送键盘命令。键盘在接收到命令后 20ms 内应予以响应，即回送一个命令响应。有的命令后还需要跟一参数（也写到该端口）。命令列表见下表。注意，如果没有另外指明，所有命令均被回送一个 0xfa 响应码(ACK)。

表 7.5 键盘命令一览表

命令码	参数	功能
0xed	有	设置/复位模式指示器。置 1 开启，0 关闭。参数字节： 位 7-3 保留全为 0; 位 2 = caps-lock 键; 位 1 = num-lock 键; 位 0 = scroll-lock 键。
0xee	无	诊断回应。键盘应回送 0xee。
0xef		保留不用。
0xf0	有	读取/设置扫描码集。参数字节等于： 0x00 - 选择当前扫描码集; 0x01 - 选择扫描码集 1(用于 PCs，PS/2 30 等); 0x02 - 选择扫描码集 2(用于 AT，PS/2，是缺省值); 0x03 - 选择扫描码集 3。
0xf1		保留不用。
0xf2	无	读取键盘标识号(读取 2 个字节)。AT 键盘返回响应码 0xfa。
0xf3	有	设置扫描码连续发送时的速率和延迟时间。参数字节的含义为： 位 7 保留为 0; 位 6-5 延时值：令 C=位 6-5，则有公式：延时值=(1+C)*250ms; 位 4-0 扫描码连续发送的速率；令 B=位 4-3；A=位 2-0，则有公式： 速率=1/((8+A)*2^B*0.00417)。 参数缺省值为 0x2c。
0xf4	无	开启键盘。
0xf5	无	禁止键盘。
0xf6	无	设置键盘默认参数。
0xf7-0xfd		保留不用。
0xfe	无	重发扫描码。当系统检测到键盘传输数据有错，则发此命令。
0xff	无	执行键盘上电复位操作，称之为基本保证测试(BAT)。操作过程为： 1. 键盘收到该命令后立刻响应发送 0xfa; 2. 键盘控制器使键盘时钟和数据线置为高电平; 3. 键盘开始执行 BAT 操作; 4. 若正常完成，则键盘发送 0xaa；否则发送 0xfd 并停止扫描。

7.7.3.3 键盘控制器命令

系统向输入缓冲(端口 0x64)写入 1 字节,即发送一键盘控制器命令。可带一参数。参数是通过写 0x60 端口发送的。

表 7.6 键盘控制器命令一览表

命令	参数	功能
0x20	无	读给键盘控制器的最后一个命令字节，放在端口 0x60 供系统读取。
0x21-0x3f	无	读取由命令低 5 比特位指定的控制器内部 RAM 中的命令。
0x60-0x7f	有	写键盘控制器命令字节。参数字节：（默认值为 0x5d） 位 7 保留为 0； 位 6 IBM PC 兼容模式(奇偶检验，转换为系统扫描码，单字节 PC 断开码)； 位 5 PC 模式（对扫描码不进行奇偶校验；不转换成系统扫描码）； 位 4 禁止键盘工作（使键盘时钟为低电平）； 位 3 禁止超越(override)，对键盘锁定转换不起作用； 位 2 系统标志；1 表示控制器工作正确； 位 1 保留为 0； 位 0 允许输出寄存器满中断。
0xaa	无	初始化键盘控制器自测试。成功返回 0x55；失败返回 0xfc。
0xab	无	初始化键盘接口测试。返回字节： 0x00 无错； 0x01 键盘时钟线为低(始终为低，低粘连)； 0x02 键盘时钟线为高； 0x03 键盘数据线为低； 0x04 键盘数据线为高；
0xac	无	诊断转储。804x 的 16 字节 RAM、输出口、输入口状态依次输出给系统。
0xad	无	禁止键盘工作（设置命令字节位 4=1）。
0xae	无	允许键盘工作（复位命令字节位 4=0）。
0xc0	无	读 804x 的输入端口 P1，并放在 0x60 供读取；
0xd0	无	读 804x 的输出端口 P2，并放在 0x60 供读取；
0xd1	有	写 804x 的输出端口 P2，原 IBM PC 使用输出端口的位 2 控制 A20 门。注意，位 0(系统复位)应该总是置位的。
0xe0	无	读测试端 T0 和 T1 的输入送输出缓冲器供系统读取。 位 1 键盘数据；位 0 键盘时钟。
0xed	有	控制 LED 的状态。置 1 开启，0 关闭。参数字节： 位 7-3 保留全为 0； 位 2 = caps-lock 键； 位 1 = num-lock 键； 位 0 = scroll-lock 键。
0xf0-0xff	无	送脉冲到输出端口。该命令序列控制输出端口 P20-23 线，参见键盘控制器逻辑示意图。欲让哪一位输出负脉冲(6 微秒)，即置该位为 0。也即该命令的低 4 位分别控制负脉冲的输出。例如，若要复位系统，则需发出命令 0xfe(P20 低)即可。

7.7.3.4 键盘扫描码

PC 机采用的均是非编码键盘。键盘上每个键都有一个位置编号，是从左到右从上到下。并且 PC XT 机与 AT 机键盘的位置码差别很大。键盘内的微处理机向系统发送的是键对应的扫描码。当键按下时，键盘输出的扫描码称为接通(make)扫描码，而该键松开时发送的则称为断开(break)扫描码。

键盘上的每个键都有一个包含在字节低 7 位（位 6-0）中相应的扫描码。在高位（位 7）表示是按键还是松开按键。位 7=0 表示刚将键按下的扫描码，位 7=1 表示键松开的扫描码。例如，如果某人刚把 ESC 键按下，则传输给系统的扫描码将是 1（1 是 ESC 键的扫描码），当该键释放时将产生 1+0x80=129 扫描码。

对于 PC、PC/XT 的标准 83 键键盘，接通扫描码与键号（键的位置码）是一样的。并用 1 字节表示。例如“A”键，键位置号是 30，接通码是扫描码是 0x1e。而其断开码是是接通扫描码加上 0x80，即 0x9e。对于 AT 机使用的 84/101/102 扩展键盘，则与 PC/XT 标准键盘区别较大。

对于某些“扩展”的键，则情况有些不同。当一个扩展键被按下时，将产生一个中断并且键盘端口将输出一个“扩展的”的扫描码前缀 0xe0，而在下一个“中断”中将给出。比如，对于 PC/XT 标准键盘，左边的控制键 ctrl 的扫描码是 29，而右边的“扩展的”控制键 ctrl 则具有一个扩展的扫描码 29。这个规则同样适合于 alt、箭头键。

另外，还有两个键的处理是非常特殊的。PrtScn 键和 Pause/Break 键。按下 PrtScn 键将会向键盘中断程序发送*2*个扩展字符，42(0x2a)和 55(0x37)，所以实际的字节序列将是 0xe0，0x2a，0xe0，0x37。但在键重复产生时将只发送扩展码 0x37。当键松开时，又重新发送两个扩展的加上 0x80 的码（0xe0，0xaa，0xe0，0xb7）。当 prtscn 键按下时，如果 shift 或 ctrl 键也按下了，则仅发送 0xe0，0x37，并且在松开时仅发送 0xe0，0xb7)。

对于 Pause/Break 键。如果你在按下该键的同时也按下了控制键，则将行如扩展键 70，而在其它情况下它将发送字符序列 0xe1，0x1d，0x45，0xe1，0x9d，0xc5。将键按下并不会产生重复的扫描码，而松开键也并不会产生任何扫描码。

因此，可以这样来看待和处理：扫描码 0xe0 意味着还有一个字符跟随其后，而扫描码 0xe1 则表示后面跟随着 2 个字符。

对于 AT 键盘的扫描码，与 PC/XT 的略有不同。当键按下时，则对应键的扫描码被送出，但当键松开时，将会发送两个字节，第一个是 0xf0，第 2 个还是相同的键扫描码。现在键盘设计者使用 8049 作为 AT 键盘的输入处理器，为了向下的兼容性将 AT 键盘发出的扫描码转换成了老式 PC/XT 标准键盘的扫描码。

AT 键盘有三种独立的扫描码集：一种是我们上面说明的(83 键映射，而增加的键有多余的 0xe0 码)，一种几乎是顺序的，还有一种却只有 1 个字节！最后一种所带来的问题是只有左 shift，caps，左 ctrl 和左 alt 键的松开码被发送。键盘的默认扫描码集是扫描码集 2，可以利用命令更改。

对于扫描码集 1 和 2，有特殊码 0xe0 和 0xe1。它们用于具有相同功能的键。比如：左控制键 ctrl 位置是 0x1d(对于 PC/XT)，则右边的控制键就是 0xe0，0x1d。这是为了与 PC/XT 程序兼容。请注意唯一使用 0xe1 的时候是当它表示临时控制键时，对此情况同时也有一个 0xe0 的版本。

The XT Scancodes

```
These are all hexadecimal:            'make' code top / 'break' below

F1 F2    `  1  2  3  4  5  6  7  8  9  0  -  =  \ BS    ESC NUML SCRL SYSR
-----   ---------------------------------------------   ------------------
3B 3C   29 02 03 04 05 06 07 08 09 0A 0B 0C 0D 2B 0E     01  45   46   **
BB BC   A9 82 83 84 85 86 87 88 89 8A 8B 8C 8D AB 8E     81  C5   C6

F3 F4   TAB   Q  W  E  R  T  Y  U  I  O  P  [  ]        Home Up  PgUp PrtSc
-----   -----------------------------------------       -------------------
3D 3E    0F   10 11 12 13 14 15 16 17 18 19 1A 1B        47  48   49   37
BD BE    8F   90 91 92 93 94 95 96 97 98 99 9A 9B        C7  C8   C9   B7

F5 F6   CNTL   A  S  D  F  G  H  J  K  L  ;  ' ENTER    Left  5  Right  -
-----   --------------------------------------------    ------------------
3F 40    1D   1E 1F 20 21 22 23 24 25 26 27 28   1C      4B  4C   4D   4A
BF C0    9D   9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8   9C      CB  CC   CD   CA

F7 F8   LSHFT    Z  X  C  V  B  N  M  ,  .  /  RSHFT    End  Dn  PgDn   +
-----   --------------------------------------------    ------------------
41 42    2A      2C 2D 2E 2F 30 31 32 33 34 35   36      4F  50   51   4E
C1 C2    AA      AC AD AE AF B0 B1 B2 B3 B4 B5   B6      CF  D0   D1   CE

F9 F10  ALT                    SPC              CAPLOCK  Ins      Del
------  --------------------------------------------    -------------
43 44    38                     39                 3A    52       53
C3 C4    B8                     B9                 BA    D2       D3
```

7.8 console.c 文件

7.8.1 功能描述

7.8.2 代码注释

列表 7.8 linux/kernel/chr_drv/console.c 程序

```
/*
 *  linux/kernel/console.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *      console.c
```

```
 *
 * This module implements the console io functions
 *      'void con_init(void)'
 *      'void con_write(struct tty_queue * queue)'
 * Hopefully this will be a rather complete VT102 implementation.
 *
 * Beeping thanks to John T Kohl.
 */
/*
 * 该模块实现控制台输入输出功能
 *      'void con_init(void)'
 *      'void con_write(struct tty_queue * queue)'
 * 希望这是一个非常完整的 VT102 实现。
 *
 * 感谢 John T Kohl 实现了蜂鸣指示。
 */

/*
 *  NOTE!!! We sometimes disable and enable interrupts for a short while
 * (to put a word in video IO), but this will work even for keyboard
 * interrupts. We know interrupts aren't enabled when getting a keyboard
 * interrupt, as we use trap-gates. Hopefully all is well.
 */
/*
 * 注意!!! 我们有时短暂地禁止和允许中断(在将一个字(word)放到视频 IO)，但即使
 * 对于键盘中断这也是可以工作的。因为我们使用陷阱门，所以我们知道在获得一个
 * 键盘中断时中断是不允许的。希望一切均正常。
 */

/*
 * Code to check for different video-cards mostly by Galen Hunt,
 * <g-hunt@ee.utah.edu>
 */
/*
 * 检测不同显示卡的代码大多数是 Galen Hunt 编写的，
 * <g-hunt@ee.utah.edu>
 */

#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/tty.h>     // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
#include <asm/io.h>        // io 头文件。定义硬件端口输入/输出宏汇编语句。
#include <asm/system.h>    // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

/*
 * These are set up by the setup-routine at boot-time:
 */
/*
 * 这些是设置子程序 setup 在引导启动系统时设置的参数：
 */

// 参见对 boot/setup.s 的注释，和 setup 程序读取并保留的参数表。
#define ORIG_X                    (*(unsigned char *)0x90000)     // 光标列号。
```

```
#define ORIG_Y                  (*(unsigned char *)0x90001)     // 光标行号。
#define ORIG_VIDEO_PAGE         (*(unsigned short *)0x90004)    // 显示页面。
#define ORIG_VIDEO_MODE         ((*(unsigned short *)0x90006) & 0xff)   // 显示模式。
#define ORIG_VIDEO_COLS         (((*(unsigned short *)0x90006) & 0xff00) >> 8) // 字符列数。
#define ORIG_VIDEO_LINES        (25)                            // 显示行数。
#define ORIG_VIDEO_EGA_AX       (*(unsigned short *)0x90008)    // [??]
#define ORIG_VIDEO_EGA_BX       (*(unsigned short *)0x9000a)    // 显示内存大小和色彩模式。
#define ORIG_VIDEO_EGA_CX       (*(unsigned short *)0x9000c)    // 显示卡特性参数。

// 定义显示器单色/彩色显示模式类型符号常数。
#define VIDEO_TYPE_MDA          0x10    /* Monochrome Text Display      */ /* 单色文本   */
#define VIDEO_TYPE_CGA          0x11    /* CGA Display                  */ /* CGA 显示器 */
#define VIDEO_TYPE_EGAM         0x20    /* EGA/VGA in Monochrome Mode   */ /* EGA/VGA 单色 */
#define VIDEO_TYPE_EGAC         0x21    /* EGA/VGA in Color Mode        */ /* EGA/VGA 彩色 */

#define NPAR 16

extern void keyboard_interrupt(void);           // 键盘中断处理程序(keyboard.S)。

static unsigned char    video_type;             /* Type of display being used   */
                                                /* 使用的显示类型 */
static unsigned long    video_num_columns;      /* Number of text columns       */
                                                /* 屏幕文本列数 */
static unsigned long    video_size_row;         /* Bytes per row                */
                                                /* 每行使用的字节数 */
static unsigned long    video_num_lines;        /* Number of test lines         */
                                                /* 屏幕文本行数 */
static unsigned char    video_page;             /* Initial video page           */
                                                /* 初始显示页面 */
static unsigned long    video_mem_start;        /* Start of video RAM           */
                                                /* 显示内存起始地址 */
static unsigned long    video_mem_end;          /* End of video RAM (sort of)   */
                                                /* 显示内存结束(末端)地址 */
static unsigned short   video_port_reg;         /* Video register select port   */
                                                /* 显示控制索引寄存器端口 */
static unsigned short   video_port_val;         /* Video register value port    */
                                                /* 显示控制数据寄存器端口 */
static unsigned short   video_erase_char;       /* Char+Attrib to erase with    */
                                                /* 擦除字符属性与字符(0x0720) */

// 以下这些变量用于屏幕卷屏操作。
static unsigned long    origin;         /* Used for EGA/VGA fast scroll */ // scr_start。
                                        /* 用于 EGA/VGA 快速滚屏 */ // 滚屏起始内存地址。
static unsigned long    scr_end;        /* Used for EGA/VGA fast scroll */
                                        /* 用于 EGA/VGA 快速滚屏 */ // 滚屏末端内存地址。
static unsigned long    pos;            // 当前光标对应的显示内存位置。
static unsigned long    x,y;            // 当前光标位置。
static unsigned long    top,bottom;     // 滚动时顶行行号；底行行号。
// state 用于标明处理 ESC 转义序列时的当前步骤。npar,par[]用于存放 ESC 序列的中间处理参数。
static unsigned long    state=0;        // ANSI 转义字符序列处理状态。
static unsigned long    npar,par[NPAR]; // ANSI 转义字符序列参数个数和参数数组。
```

```
static unsigned long    ques=0;
static unsigned char    attr=0x07;          // 字符属性(黑底白字)。

static void sysbeep(void);                  // 系统蜂鸣函数。

/*
 * this is what the terminal answers to a ESC-Z or csi0c
 * query (= vt100 response).
 */
/*
 * 下面是终端回应 ESC-Z 或 csi0c 请求的应答(=vt100 响应)。
 */
// csi - 控制序列引导码(Control Sequence Introducer)。
#define RESPONSE "\033[?1;2c"

/* NOTE! gotoxy thinks x==video_num_columns is ok */
/* 注意！gotoxy 函数认为 x==video_num_columns，这是正确的 */
//// 跟踪光标当前位置。
// 参数：new_x - 光标所在列号；new_y - 光标所在行号。
// 更新当前光标位置变量 x,y，并修正 pos 指向光标在显示内存中的对应位置。
static inline void gotoxy(unsigned int new_x,unsigned int new_y)
{
// 如果输入的光标行号超出显示器列数，或者光标行号超出显示的最大行数，则退出。
        if (new_x > video_num_columns || new_y >= video_num_lines)
                return;
// 更新当前光标变量；更新光标位置对应的在显示内存中位置变量 pos。
        x=new_x;
        y=new_y;
        pos=origin + y*video_size_row + (x<<1);
}

//// 设置滚屏起始显示内存地址。
static inline void set_origin(void)
{
        cli();
// 首先选择显示控制数据寄存器 r12，然后写入卷屏起始地址高字节。向右移动 9 位，表示向右移动
// 8 位，再除以 2(2 字节代表屏幕上 1 字符)。是相对于默认显示内存操作的。
        outb_p(12, video_port_reg);
        outb_p(0xff&((origin-video_mem_start)>>9), video_port_val);
// 再选择显示控制数据寄存器 r13，然后写入卷屏起始地址底字节。向右移动 1 位表示除以 2。
        outb_p(13, video_port_reg);
        outb_p(0xff&((origin-video_mem_start)>>1), video_port_val);
        sti();
}

//// 向上卷动一行（屏幕窗口向下移动）。
// 将屏幕窗口向下移动一行。参见程序列表后说明。
static void scrup(void)
{
// 如果显示类型是 EGA，则执行以下操作。
        if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
        {
// 如果移动起始行 top=0，移动最底行 bottom=video_num_lines=25，则表示整屏窗口向下移动。
```

```
                if (!top && bottom == video_num_lines) {
// 调整屏幕显示对应内存的起始位置指针 origin 为向下移一行屏幕字符对应的内存位置，同时也调整
// 当前光标对应的内存位置以及屏幕末行末端字符指针 scr_end 的位置。
                        origin += video_size_row;
                        pos += video_size_row;
                        scr_end += video_size_row;
// 如果屏幕末端最后一个显示字符所对应的显示内存指针 scr_end 超出了实际显示内存的末端，则将
// 屏幕内容内存数据移动到显示内存的起始位置 video_mem_start 处，并在出现的新行上填入空格字符。
                        if (scr_end > video_mem_end) {
// %0 - eax(擦除字符+属性)；%1 - ecx((显示器字符行数-1)所对应的字符数/2，是以长字移动)；
// %2 - edi(显示内存起始位置 video_mem_start)；%3 - esi(屏幕内容对应的内存起始位置 origin)。
// 移动方向：[edi]➔[esi]，移动 ecx 个长字。
                                __asm__("cld\n\t"          // 清方向位。
                                        "rep\n\t"          // 重复操作，将当前屏幕内存数据
                                        "movsl\n\t"        // 移动到显示内存起始处。
                                        "movl _video_num_columns,%1\n\t" // ecx=1 行字符数。
                                        "rep\n\t"          // 在新行上填入空格字符。
                                        "stosw"
                                        ::"a" (video_erase_char),
                                        "c" ((video_num_lines-1)*video_num_columns>>1),
                                        "D" (video_mem_start),
                                        "S" (origin)
                                        :"cx","di","si");
// 根据屏幕内存数据移动后的情况，重新调整当前屏幕对应内存的起始指针、光标位置指针和屏幕末端
// 对应内存指针 scr_end。
                                scr_end -= origin-video_mem_start;
                                pos -= origin-video_mem_start;
                                origin = video_mem_start;
                        } else {
// 如果调整后的屏幕末端对应的内存指针 scr_end 没有超出显示内存的末端 video_mem_end，则只需在
// 新行上填入擦除字符(空格字符)。
// %0 - eax(擦除字符+属性)；%1 - ecx(显示器字符行数)；%2 - edi(屏幕对应内存最后一行开始处)；
                                __asm__("cld\n\t"       // 清方向位。
                                        "rep\n\t"       // 重复操作，在新出现行上
                                        "stosw"         // 填入擦除字符(空格字符)。
                                        ::"a" (video_erase_char),
                                        "c" (video_num_columns),
                                        "D" (scr_end-video_size_row)
                                        :"cx","di");
                        }
// 向显示控制器中写入新的屏幕内容对应的内存起始位置值。
                        set_origin();
// 否则表示不是整屏移动。也即表示从指定行 top 开始的所有行向上移动 1 行(删除 1 行)。此时直接
// 将屏幕从指定行 top 到屏幕末端所有行对应的显示内存数据向上移动 1 行，并在新出现的行上填入擦
// 除字符。
// %0-eax(擦除字符+属性)；%1-ecx(top 行下 1 行开始到屏幕末行的行数所对应的内存长字数)；
// %2-edi(top 行所处的内存位置)；%3-esi(top+1 行所处的内存位置)。
                } else {
                        __asm__("cld\n\t"              // 清方向位。
                                "rep\n\t"              // 循环操作，将 top+1 到 bottom 行
                                "movsl\n\t"            // 所对应的内存块移到 top 行开始处。
                                "movl _video_num_columns,%%ecx\n\t"  // ecx = 1 行字符数。
                                "rep\n\t"              // 在新行上填入擦除字符。
```

```
                                        "stosw"
                                        :: "a" (video_erase_char),
                                        "c" ((bottom-top-1)*video_num_columns>>1),
                                        "D" (origin+video_size_row*top),
                                        "S" (origin+video_size_row*(top+1))
                                        :"cx","di","si");
                        }
                }
// 如果显示类型不是 EGA(是 MDA)，则执行下面移动操作。因为 MDA 显示控制卡会自动调整超出显示范围
// 的情况，也即会自动翻卷指针，所以这里不对屏幕内容对应内存超出显示内存的情况单独处理。处理
// 方法与 EGA 非整屏移动情况完全一样。
        else            /* Not EGA/VGA */
        {
                __asm__("cld\n\t"
                        "rep\n\t"
                        "movsl\n\t"
                        "movl _video_num_columns,%%ecx\n\t"
                        "rep\n\t"
                        "stosw"
                        :: "a" (video_erase_char),
                        "c" ((bottom-top-1)*video_num_columns>>1),
                        "D" (origin+video_size_row*top),
                        "S" (origin+video_size_row*(top+1))
                        :"cx","di","si");
        }
}

//// 向下卷动一行（屏幕窗口向上移动）。
// 将屏幕窗口向上移动一行，屏幕显示的内容向下移动 1 行，在被移动开始行的上方出现一新行。参见
// 程序列表后说明。处理方法与 scrup()相似，只是为了在移动显示内存数据时不出现数据覆盖错误情
// 况，复制是以反方向进行的，也即从屏幕倒数第 2 行的最后一个字符开始复制
static void scrdown(void)
{
// 如果显示类型是 EGA，则执行下列操作。
// [??好象 if 和 else 的操作完全一样啊!为什么还要分别处理呢？难道与任务切换有关？]
        if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
        {
// %0-eax(擦除字符+属性)；%1-ecx(top 行开始到屏幕末行-1 行的行数所对应的内存长字数)；
// %2-edi(屏幕右下角最后一个长字位置)；%3-esi(屏幕倒数第 2 行最后一个长字位置)。
// 移动方向：[esi]➔[edi]，移动 ecx 个长字。
                __asm__("std\n\t"               // 置方向位。
                        "rep\n\t"               // 重复操作，向下移动从 top 行到 bottom-1 行
                        "movsl\n\t"             // 对应的内存数据。
                        "addl $2,%%edi\n\t"     /* %edi has been decremented by 4 */
                                                /* %edi 已经减 4，因为也是方向填擦除字符 */
                        "movl _video_num_columns,%%ecx\n\t" // 置 ecx=1 行字符数。
                        "rep\n\t"               // 将擦除字符填入上方新行中。
                        "stosw"
                        :: "a" (video_erase_char),
                        "c" ((bottom-top-1)*video_num_columns>>1),
                        "D" (origin+video_size_row*bottom-4),
                        "S" (origin+video_size_row*(bottom-1)-4)
```

```
                                :"ax","cx","di","si");
                }
// 如果不是 EGA 显示类型，则执行以下操作（目前与上面完全一样）。
                else                    /* Not EGA/VGA */
                {
                        __asm__("std\n\t"
                                "rep\n\t"
                                "movsl\n\t"
                                "addl $2,%%edi\n\t"     /* %edi has been decremented by 4 */
                                "movl _video_num_columns,%%ecx\n\t"
                                "rep\n\t"
                                "stosw"
                                ::"a" (video_erase_char),
                                "c" ((bottom-top-1)*video_num_columns>>1),
                                "D" (origin+video_size_row*bottom-4),
                                "S" (origin+video_size_row*(bottom-1)-4)
                                :"ax","cx","di","si");
                }
}

//// 光标位置下移一行(lf - line feed 换行)。
static void lf(void)
{
// 如果光标没有处在倒数第 2 行之后，则直接修改光标当前行变量 y++，并调整光标对应显示内存位置
// pos(加上屏幕一行字符所对应的内存长度)。
        if (y+1<bottom) {
                y++;
                pos += video_size_row;
                return;
        }
// 否则需要将屏幕内容上移一行。
        scrup();
}

//// 光标上移一行(ri - reverse line feed 反向换行)。
static void ri(void)
{
// 如果光标不在第 1 行上，则直接修改光标当前行标量 y--，并调整光标对应显示内存位置 pos，减去
// 屏幕上一行字符所对应的内存长度字节数。
        if (y>top) {
                y--;
                pos -= video_size_row;
                return;
        }
// 否则需要将屏幕内容下移一行。
        scrdown();
}

// 光标回到第 1 列(0 列)左端(cr - carriage return 回车)。
static void cr(void)
{
// 光标所在的列号*2 即 0 列到光标所在列对应的内存字节长度。
        pos -= x<<1;
```

```
        x=0;
}

// 擦除光标前一字符(用空格替代)(del - delete 删除)。
static void del(void)
{
// 如果光标没有处在 0 列，则将光标对应内存位置指针 pos 后退 2 字节(对应屏幕上一个字符)，然后
// 将当前光标变量列值减 1，并将光标所在位置字符擦除。
        if (x) {
                pos -= 2;
                x--;
                *(unsigned short *)pos = video_erase_char;
        }
}

//// 删除屏幕上与光标位置相关的部分，以屏幕为单位。csi - 控制序列引导码(Control Sequence
// Introducer)。
// ANSI 转义序列：'ESC [sJ'(s = 0 删除光标到屏幕底端；1 删除屏幕开始到光标处；2 整屏删除)。
// 参数：par - 对应上面 s。
static void csi_J(int par)
{
        long count __asm__("cx");       // 设为寄存器变量。
        long start __asm__("di");

// 首先根据三种情况分别设置需要删除的字符数和删除开始的显示内存位置。
        switch (par) {
                case 0: /* erase from cursor to end of display */  /* 擦除光标到屏幕底端 */
                        count = (scr_end-pos)>>1;
                        start = pos;
                        break;
                case 1: /* erase from start to cursor */  /* 删除从屏幕开始到光标处的字符 */
                        count = (pos-origin)>>1;
                        start = origin;
                        break;
                case 2: /* erase whole display */    /* 删除整个屏幕上的字符 */
                        count = video_num_columns * video_num_lines;
                        start = origin;
                        break;
                default:
                        return;
        }
// 然后使用擦除字符填写删除字符的地方。
// %0 - ecx(要删除的字符数 count)；%1 - edi(删除操作开始地址)；%2 - eax（填入的擦除字符）。
        __asm__("cld\n\t"
                "rep\n\t"
                "stosw\n\t"
                ::"c" (count),
                "D" (start),"a" (video_erase_char)
                :"cx","di");
}

//// 删除行内与光标位置相关的部分，以一行为单位。
// ANSI 转义字符序列：'ESC [sK'(s = 0 删除到行尾；1 从开始删除；2 整行都删除)。
```

```
static void csi_K(int par)
{
        long count __asm__("cx");         // 设置寄存器变量。
        long start __asm__("di");

// 首先根据三种情况分别设置需要删除的字符数和删除开始的显示内存位置。
        switch (par) {
                case 0: /* erase from cursor to end of line */  /* 删除光标到行尾字符 */
                        if (x>=video_num_columns)
                                return;
                        count = video_num_columns-x;
                        start = pos;
                        break;
                case 1: /* erase from start of line to cursor */  /* 删除从行开始到光标处 */
                        start = pos - (x<<1);
                        count = (x<video_num_columns)?x:video_num_columns;
                        break;
                case 2: /* erase whole line */  /* 将整行字符全删除 */
                        start = pos - (x<<1);
                        count = video_num_columns;
                        break;
                default:
                        return;
        }
// 然后使用擦除字符填写删除字符的地方。
// %0 - ecx(要删除的字符数 count)；%1 - edi(删除操作开始地址)；%2 - eax（填入的擦除字符）。
        __asm__("cld\n\t"
                "rep\n\t"
                "stosw\n\t"
                ::"c" (count),
                "D" (start),"a" (video_erase_char)
                :"cx","di");
}

//// 允许翻译(重显)（允许重新设置字符显示方式，比如加粗、加下划线、闪烁、反显等）。
// ANSI 转义字符序列：'ESC [nm'。n = 0 正常显示；1 加粗；4 加下划线；7 反显；27 正常显示。
void csi_m(void)
{
        int i;

        for (i=0;i<=npar;i++)
                switch (par[i]) {
                        case 0:attr=0x07;break;
                        case 1:attr=0x0f;break;
                        case 4:attr=0x0f;break;
                        case 7:attr=0x70;break;
                        case 27:attr=0x07;break;
                }
}

//// 根据设置显示光标。
// 根据显示内存光标对应位置 pos，设置显示控制器光标的显示位置。
static inline void set_cursor(void)
```

```
{
        cli();
// 首先使用索引寄存器端口选择显示控制数据寄存器 r14(光标当前显示位置高字节)，然后写入光标
// 当前位置高字节(向右移动 9 位表示高字节移到低字节再除以 2)。是相对于默认显示内存操作的。
        outb_p(14, video_port_reg);
        outb_p(0xff&((pos-video_mem_start)>>9), video_port_val);
// 再使用索引寄存器选择 r15，并将光标当前位置低字节写入其中。
        outb_p(15, video_port_reg);
        outb_p(0xff&((pos-video_mem_start)>>1), video_port_val);
        sti();
}

//// 发送对终端 VT100 的响应序列。
// 将响应序列放入读缓冲队列中。
static void respond(struct tty_struct * tty)
{
        char * p = RESPONSE;

        cli();                  // 关中断。
        while (*p) {            // 将字符序列放入写队列。
                PUTCH(*p,tty->read_q);
                p++;
        }
        sti();                  // 开中断。
        copy_to_cooked(tty);    // 转换成规范模式(放入辅助队列中)。
}

//// 在光标处插入一空格字符。
static void insert_char(void)
{
        int i=x;
        unsigned short tmp, old = video_erase_char;
        unsigned short * p = (unsigned short *) pos;

// 光标开始的所有字符右移一格，并将擦除字符插入在光标所在处。
// 若一行上都有字符的话，则行最后一个字符将不会更动☺?
        while (i++<video_num_columns) {
                tmp=*p;
                *p=old;
                old=tmp;
                p++;
        }
}

//// 在光标处插入一行（则光标将处在新的空行上）。
// 将屏幕从光标所在行到屏幕底向下卷动一行。
static void insert_line(void)
{
        int oldtop,oldbottom;

        oldtop=top;                     // 保存原 top，bottom 值。
        oldbottom=bottom;
        top=y;                          // 设置屏幕卷动开始行。
```

```
        bottom = video_num_lines;       // 设置屏幕卷动最后行。
        scrdown();                      // 从光标开始处，屏幕内容向下滚动一行。
        top=oldtop;                     // 恢复原 top，bottom 值。
        bottom=oldbottom;
}

//// 删除光标处的一个字符。
static void delete_char(void)
{
        int i;
        unsigned short * p = (unsigned short *) pos;

// 如果光标超出屏幕最右列，则返回。
        if (x>=video_num_columns)
                return;
// 从光标右一个字符开始到行末所有字符左移一格。
        i = x;
        while (++i < video_num_columns) {
                *p = *(p+1);
                p++;
        }
// 最后一个字符处填入擦除字符(空格字符)。
        *p = video_erase_char;
}

//// 删除光标所在行。
// 从光标所在行开始屏幕内容上卷一行。
static void delete_line(void)
{
        int oldtop,oldbottom;

        oldtop=top;                     // 保存原 top，bottom 值。
        oldbottom=bottom;
        top=y;                          // 设置屏幕卷动开始行。
        bottom = video_num_lines;       // 设置屏幕卷动最后行。
        scrup();                        // 从光标开始处，屏幕内容向上滚动一行。
        top=oldtop;                     // 恢复原 top，bottom 值。
        bottom=oldbottom;
}

//// 在光标处插入 nr 个字符。
// ANSI 转义字符序列：'ESC [n@ '。
// 参数 nr = 上面 n。
static void csi_at(unsigned int nr)
{
// 如果插入的字符数大于一行字符数，则截为一行字符数；若插入字符数 nr 为 0，则插入 1 个字符。
        if (nr > video_num_columns)
                nr = video_num_columns;
        else if (!nr)
                nr = 1;
// 循环插入指定的字符数。
        while (nr--)
                insert_char();
```

```
}

//// 在光标位置处插入 nr 行。
// ANSI 转义字符序列'ESC [nL'。
static void csi_L(unsigned int nr)
{
// 如果插入的行数大于屏幕最多行数，则截为屏幕显示行数；若插入行数 nr 为 0，则插入 1 行。
        if (nr > video_num_lines)
                nr = video_num_lines;
        else if (!nr)
                nr = 1;
// 循环插入指定行数 nr。
        while (nr--)
                insert_line();
}

//// 删除光标处的 nr 个字符。
// ANSI 转义序列：'ESC [nP'。
static void csi_P(unsigned int nr)
{
// 如果删除的字符数大于一行字符数，则截为一行字符数；若删除字符数 nr 为 0，则删除 1 个字符。
        if (nr > video_num_columns)
                nr = video_num_columns;
        else if (!nr)
                nr = 1;
// 循环删除指定字符数 nr。
        while (nr--)
                delete_char();
}

//// 删除光标处的 nr 行。
// ANSI 转义序列：'ESC [nM'。
static void csi_M(unsigned int nr)
{
// 如果删除的行数大于屏幕最多行数，则截为屏幕显示行数；若删除的行数 nr 为 0，则删除 1 行。
        if (nr > video_num_lines)
                nr = video_num_lines;
        else if (!nr)
                nr=1;
// 循环删除指定行数 nr。
        while (nr--)
                delete_line();
}

static int saved_x=0;           // 保存的光标列号。
static int saved_y=0;           // 保存的光标行号。

//// 保存当前光标位置。
static void save_cur(void)
{
        saved_x=x;
        saved_y=y;
}
```

```
//// 恢复保存的光标位置。
static void restore_cur(void)
{
        gotoxy(saved_x, saved_y);
}

//// 控制台写函数。
// 从终端对应的 tty 写缓冲队列中取字符，并显示在屏幕上。
void con_write(struct tty_struct * tty)
{
        int nr;
        char c;

// 首先取得写缓冲队列中现有字符数 nr，然后针对每个字符进行处理。
        nr = CHARS(tty->write_q);
        while (nr--) {
// 从写队列中取一字符 c，根据前面所处理字符的状态 state 分别处理。状态之间的转换关系为：
// state = 0：初始状态；或者原是状态 4；或者原是状态 1，但字符不是'['；
//         1：原是状态 0，并且字符是转义字符 ESC(0x1b = 033 = 27)；
//         2：原是状态 1，并且字符是'['；
//         3：原是状态 2；或者原是状态 3，并且字符是';'或数字。
//         4：原是状态 3，并且字符不是';'或数字；
                GETCH(tty->write_q,c);
                switch(state) {
                        case 0:
// 如果字符不是控制字符(c>31)，并且也不是扩展字符(c<127)，则
                                if (c>31 && c<127) {
// 若当前光标处在行末端或末端以外，则将光标移到下行头列。并调整光标位置对应的内存指针 pos。
                                        if (x>=video_num_columns) {
                                                x -= video_num_columns;
                                                pos -= video_size_row;
                                                lf();
                                        }
// 将字符 c 写到显示内存中 pos 处，并将光标右移 1 列，同时也将 pos 对应地移动 2 个字节。
                                        __asm__("movb _attr,%%ah\n\t"
                                                "movw %%ax,%1\n\t"
                                                ::"a" (c),"m" (*(short *)pos)
                                                :"ax");
                                        pos += 2;
                                        x++;
// 如果字符 c 是转义字符 ESC，则转换状态 state 到 1。
                                } else if (c==27)
                                        state=1;
// 如果字符 c 是换行符(10)，或是垂直制表符 VT(11)，或者是换页符 FF(12)，则移动光标到下一行。
                                else if (c==10 || c==11 || c==12)
                                        lf();
// 如果字符 c 是回车符 CR(13)，则将光标移动到头列(0 列)。
                                else if (c==13)
                                        cr();
// 如果字符 c 是 DEL(127)，则将光标右边一字符擦除(用空格字符替代)，并将光标移到被擦除位置。
                                else if (c==ERASE_CHAR(tty))
                                        del();
```

```
// 如果字符 c 是 BS(backspace,8)，则将光标右移 1 格，并相应调整光标对应内存位置指针 pos。
                                } else if (c==8) {
                                        if (x) {
                                                x--;
                                                pos -= 2;
                                        }
// 如果字符 c 是水平制表符 TAB(9)，则将光标移到 8 的倍数列上。若此时光标列数超出屏幕最大列数，
// 则将光标移到下一行上。
                                } else if (c==9) {
                                        c=8-(x&7);
                                        x += c;
                                        pos += c<<1;
                                        if (x>video_num_columns) {
                                                x -= video_num_columns;
                                                pos -= video_size_row;
                                                lf();
                                        }
                                        c=9;
// 如果字符 c 是响铃符 BEL(7)，则调用蜂鸣函数，是扬声器发声。
                                } else if (c==7)
                                        sysbeep();
                                break;
// 如果原状态是 0，并且字符是转义字符 ESC(0x1b = 033 = 27)，则转到状态 1 处理。
                        case 1:
                                state=0;
// 如果字符 c 是'['，则将状态 state 转到 2。
                                if (c=='[')
                                        state=2;
// 如果字符 c 是'E'，则光标移到下一行开始处(0 列)。
                                else if (c=='E')
                                        gotoxy(0,y+1);
// 如果字符 c 是'M'，则光标上移一行。
                                else if (c=='M')
                                        ri();
// 如果字符 c 是'D'，则光标下移一行。
                                else if (c=='D')
                                        lf();
// 如果字符 c 是'Z'，则发送终端应答字符序列。
                                else if (c=='Z')
                                        respond(tty);
// 如果字符 c 是'7'，则保存当前光标位置。注意这里代码写错！应该是(c=='7')。
                                else if (x=='7')
                                        save_cur();
// 如果字符 c 是'8'，则恢复到原保存的光标位置。注意这里代码写错！应该是(c=='8')。
                                else if (x=='8')
                                        restore_cur();
                                break;
// 如果原状态是 1，并且上一字符是'['，则转到状态 2 来处理。
                        case 2:
// 首先对 ESC 转义字符序列参数使用的处理数组 par[]清零，索引变量 npar 指向首项，并且设置状态
// 为 3。若此时字符不是'?'，则直接转到状态 3 去处理，否则去读一字符，再到状态 3 处理代码处。
                                for(npar=0;npar<NPAR;npar++)
                                        par[npar]=0;
```

```
                                        npar=0;
                                        state=3;
                                        if (ques=(c=='?'))
                                                break;
// 如果原来是状态 2；或者原来就是状态 3，但原字符是';'或数字，则在下面处理。
                                case 3:
// 如果字符 c 是分号';'，并且数组 par 未满，则索引值加 1。
                                        if (c==';' && npar<NPAR-1) {
                                                npar++;
                                                break;
// 如果字符 c 是数字字符'0'-'9'，则将该字符转换成数值并与 npar 所索引的项组成 10 进制数。
                                        } else if (c>='' && c<='9') {
                                                par[npar]=10*par[npar]+c-'';
                                                break;
// 否则转到状态 4。
                                        } else state=4;
// 如果原状态是状态 3，并且字符不是';'或数字，则转到状态 4 处理。首先复位状态 state=0。
                                case 4:
                                        state=0;
                                        switch(c) {
// 如果字符 c 是'G'或'`'，则 par[]中第一个参数代表列号。若列号不为零，则将光标右移一格。
                                                case 'G': case '`':
                                                        if (par[0]) par[0]--;
                                                        gotoxy(par[0],y);
                                                        break;
// 如果字符 c 是'A'，则第一个参数代表光标上移的行数。若参数为 0 则上移一行。
                                                case 'A':
                                                        if (!par[0]) par[0]++;
                                                        gotoxy(x,y-par[0]);
                                                        break;
// 如果字符 c 是'B'或'e'，则第一个参数代表光标下移的行数。若参数为 0 则下移一行。
                                                case 'B': case 'e':
                                                        if (!par[0]) par[0]++;
                                                        gotoxy(x,y+par[0]);
                                                        break;
// 如果字符 c 是'C'或'a'，则第一个参数代表光标右移的格数。若参数为 0 则右移一格。
                                                case 'C': case 'a':
                                                        if (!par[0]) par[0]++;
                                                        gotoxy(x+par[0],y);
                                                        break;
// 如果字符 c 是'D'，则第一个参数代表光标左移的格数。若参数为 0 则左移一格。
                                                case 'D':
                                                        if (!par[0]) par[0]++;
                                                        gotoxy(x-par[0],y);
                                                        break;
// 如果字符 c 是'E'，则第一个参数代表光标向下移动的行数，并回到 0 列。若参数为 0 则下移一行。
                                                case 'E':
                                                        if (!par[0]) par[0]++;
                                                        gotoxy(0,y+par[0]);
                                                        break;
// 如果字符 c 是'F'，则第一个参数代表光标向上移动的行数，并回到 0 列。若参数为 0 则上移一行。
                                                case 'F':
                                                        if (!par[0]) par[0]++;
```

```
                                        gotoxy(0,y-par[0]);
                                        break;
// 如果字符 c 是'd'，则第一个参数代表光标所需在的行号(从 0 计数)。
                                case 'd':
                                        if (par[0]) par[0]--;
                                        gotoxy(x,par[0]);
                                        break;
// 如果字符 c 是'H'或'f'，则第一个参数代表光标移到的行号，第二个参数代表光标移到的列号。
                                case 'H': case 'f':
                                        if (par[0]) par[0]--;
                                        if (par[1]) par[1]--;
                                        gotoxy(par[1],par[0]);
                                        break;
// 如果字符 c 是'J'，则第一个参数代表以光标所处位置清屏的方式：
// ANSI 转义序列：'ESC [sJ'(s = 0 删除光标到屏幕底端；1 删除屏幕开始到光标处；2 整屏删除)。
                                case 'J':
                                        csi_J(par[0]);
                                        break;
// 如果字符 c 是'K'，则第一个参数代表以光标所在位置对行中字符进行删除处理的方式。
// ANSI 转义字符序列：'ESC [sK'(s = 0 删除到行尾；1 从开始删除；2 整行都删除)。
                                case 'K':
                                        csi_K(par[0]);
                                        break;
// 如果字符 c 是'L'，表示在光标位置处插入 n 行(ANSI 转义字符序列'ESC [nL')。
                                case 'L':
                                        csi_L(par[0]);
                                        break;
// 如果字符 c 是'M'，表示在光标位置处删除 n 行(ANSI 转义字符序列'ESC [nM')。
                                case 'M':
                                        csi_M(par[0]);
                                        break;
// 如果字符 c 是'P'，表示在光标位置处删除 n 个字符(ANSI 转义字符序列'ESC [nP')。
                                case 'P':
                                        csi_P(par[0]);
                                        break;
// 如果字符 c 是'@'，表示在光标位置处插入 n 个字符(ANSI 转义字符序列'ESC [n@')。
                                case '@':
                                        csi_at(par[0]);
                                        break;
// 如果字符 c 是'm'，表示改变光标处字符的显示属性，比如加粗、加下划线、闪烁、反显等。
// ANSI 转义字符序列：'ESC [nm'。n = 0 正常显示；1 加粗；4 加下划线；7 反显；27 正常显示。
                                case 'm':
                                        csi_m();
                                        break;
// 如果字符 c 是'r'，则表示用两个参数设置滚屏的起始行号和终止行号。
                                case 'r':
                                        if (par[0]) par[0]--;
                                        if (!par[1]) par[1] = video_num_lines;
                                        if (par[0] < par[1] &&
                                            par[1] <= video_num_lines) {
                                                top=par[0];
                                                bottom=par[1];
                                        }
```

```
                                        break;
// 如果字符 c 是's'，则表示保存当前光标所在位置。
                                case 's':
                                        save_cur();
                                        break;
// 如果字符 c 是'u'，则表示恢复光标到原保存的位置处。
                                case 'u':
                                        restore_cur();
                                        break;
                        }
                }
        }
// 最后根据上面设置的光标位置，向显示控制器发送光标显示位置。
        set_cursor();
}

/*
 *  void con_init(void);
 *
 * This routine initalizes console interrupts, and does nothing
 * else. If you want the screen to clear, call tty_write with
 * the appropriate escape-sequece.
 *
 * Reads the information preserved by setup.s to determine the current display
 * type and sets everything accordingly.
 */
/*
 * void con_init(void);
 * 这个子程序初始化控制台中断，其它什么都不做。如果你想让屏幕干净的话，就使用
 * 适当的转义字符序列调用 tty_write()函数。
 *
 * 读取 setup.s 程序保存的信息，用以确定当前显示器类型，并且设置所有相关参数。
 */
void con_init(void)
{
        register unsigned char a;
        char *display_desc = "????";
        char *display_ptr;

        video_num_columns = ORIG_VIDEO_COLS;          // 显示器显示字符列数。
        video_size_row = video_num_columns * 2;       // 每行需使用字节数。
        video_num_lines = ORIG_VIDEO_LINES;           // 显示器显示字符行数。
        video_page = ORIG_VIDEO_PAGE;                 // 当前显示页面。
        video_erase_char = 0x0720;                    // 擦除字符(0x20 显示字符， 0x07 是属性)。

// 如果原始显示模式等于 7，则表示是单色显示器。
        if (ORIG_VIDEO_MODE == 7)                     /* Is this a monochrome display? */
        {
                video_mem_start = 0xb0000;            // 设置单显映象内存起始地址。
                video_port_reg = 0x3b4;               // 设置单显索引寄存器端口。
                video_port_val = 0x3b5;               // 设置单显数据寄存器端口。
// 根据 BIOS 中断 int 0x10 功能 0x12 获得的显示模式信息，判断显示卡单色显示卡还是彩色显示卡。
// 如果使用上述中断功能所得到的 BX 寄存器返回值不等于 0x10，则说明是 EGA 卡。因此初始
```

```
// 显示类型为 EGA 单色；所使用映象内存末端地址为 0xb8000；并置显示器描述字符串为'EGAm'。
// 在系统初始化期间显示器描述字符串将显示在屏幕的右上角。
                if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
                {
                        video_type = VIDEO_TYPE_EGAM;   // 设置显示类型(EGA 单色)。
                        video_mem_end = 0xb8000;        // 设置显示内存末端地址。
                        display_desc = "EGAm";          // 设置显示描述字符串。
                }
// 如果 BX 寄存器的值等于 0x10，则说明是单色显示卡 MDA。则设置相应参数。
                else
                {
                        video_type = VIDEO_TYPE_MDA;    // 设置显示类型(MDA 单色)。
                        video_mem_end   = 0xb2000;      // 设置显示内存末端地址。
                        display_desc = "*MDA";          // 设置显示描述字符串。
                }
        }
// 如果显示模式不为 7，则为彩色模式。此时所用的显示内存起始地址为 0xb800；显示控制索引寄存
// 器端口地址为 0x3d4；数据寄存器端口地址为 0x3d5。
        else                                            /* If not, it is color. */
        {
                video_mem_start = 0xb8000;              // 显示内存起始地址。
                video_port_reg  = 0x3d4;                // 设置彩色显示索引寄存器端口。
                video_port_val  = 0x3d5;                // 设置彩色显示数据寄存器端口。
// 再判断显示卡类别。如果 BX 不等于 0x10，则说明是 EGA 显示卡。
                if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
                {
                        video_type = VIDEO_TYPE_EGAC;   // 设置显示类型(EGA 彩色)。
                        video_mem_end = 0xbc000;        // 设置显示内存末端地址。
                        display_desc = "EGAc";          // 设置显示描述字符串。
                }
// 如果 BX 寄存器的值等于 0x10，则说明是 CGA 显示卡。则设置相应参数。
                else
                {
                        video_type = VIDEO_TYPE_CGA;    // 设置显示类型(CGA)。
                        video_mem_end = 0xba000;        // 设置显示内存末端地址。
                        display_desc = "*CGA";          // 设置显示描述字符串。
                }
        }

        /* Let the user known what kind of display driver we are using */
        /* 让用户知道我们正在使用哪一类显示驱动程序 */

// 在屏幕的右上角显示显示描述字符串。采用的方法是直接将字符串写到显示内存的相应位置处。
// 首先将显示指针 display_ptr 指到屏幕第一行右端差 4 个字符处(每个字符需 2 个字节，因此减 8)。
        display_ptr = ((char *)video_mem_start) + video_size_row - 8;
// 然后循环复制字符串中的字符，并且每复制一个字符都空开一个属性字节。
        while (*display_desc)
        {
                *display_ptr++ = *display_desc++;       // 复制字符。
                display_ptr++;                          // 空开属性字节位置。
        }

        /* Initialize the variables used for scrolling (mostly EGA/VGA) */
```

```
        /* 初始化用于滚屏的变量(主要用于 EGA/VGA) */

        origin  = video_mem_start;                  // 滚屏起始显示内存地址。
        scr_end = video_mem_start + video_num_lines * video_size_row;  // 滚屏结束内存地址。
        top     = 0;                                // 最顶行号。
        bottom  = video_num_lines;                  // 最底行号。

        gotoxy(ORIG_X,ORIG_Y);                      // 初始化光标位置 x,y 和对应的内存位置 pos。
        set_trap_gate(0x21,&keyboard_interrupt);  // 设置键盘中断陷阱门。
        outb_p(inb_p(0x21)&0xfd,0x21);              // 取消 8259A 中对键盘中断的屏蔽,允许 IRQ1。
        a=inb_p(0x61);                              // 延迟读取键盘端口 0x61(8255A 端口 PB)。
        outb_p(a|0x80,0x61);                        // 设置禁止键盘工作(位 7 置位),
        outb(a,0x61);                               // 再允许键盘工作,用以复位键盘操作。
}
/* from bsd-net-2: */

//// 停止蜂鸣。
// 复位 8255A PB 端口的位 1 和位 0。
void sysbeepstop(void)
{
        /* disable counter 2 */  /* 禁止定时器 2 */
        outb(inb_p(0x61)&0xFC, 0x61);
}

int beepcount = 0;

// 开通蜂鸣。
// 8255A 芯片 PB 端口的位 1 用作扬声器的开门信号;位 0 用作 8253 定时器 2 的门信号,该定时器的
// 输出脉冲送往扬声器,作为扬声器发声的频率。因此要使扬声器蜂鸣,需要两步:首先开启 PB 端口
// 位 1 和位 0(置位),然后设置定时器发送一定的定时频率即可。
static void sysbeep(void)
{
        /* enable counter 2 */  /* 开启定时器 2 */
        outb_p(inb_p(0x61)|3, 0x61);
        /* set command for counter 2, 2 byte write */  /* 送设置定时器 2 命令 */
        outb_p(0xB6, 0x43);
        /* send 0x637 for 750 HZ */  /* 设置频率为 750HZ,因此送定时值 0x637 */
        outb_p(0x37, 0x42);
        outb(0x06, 0x42);
        /* 1/8 second */  /* 蜂鸣时间为 1/8 秒 */
        beepcount = HZ/8;
}
```

7.8.3 其它信息

7.8.3.1 显示控制卡编程

这里仅给出和说明兼容显示卡端口的说明。描述了 MDA、CGA、EGA 和 VGA 显示控制卡的通用编程端口,这些端口都是与 CGA 使用的 MC6845 芯片兼容,其名称和用途见下面列表。其中以 CGA/EGA/VGA 的端口(0x3d0-0x3df)为例进行说明,MDA 的端口是 0x3b0 - 0x3bf。

对显示控制卡进行编程的基本步骤是：首先写显示卡的索引寄存器，选择要进行设置的显示控制内部寄存器之一(r0-r17)，然后将参数写到其数据寄存器端口。也即显示卡的数据寄存器端口每次只能对显示卡中的一个内部寄存器进行操作。

表 7.7 CGA 端口寄存器名称及作用

端口	读/写	名称和用途
0x3d4	写	CRT(6845)索引寄存器。用于选择通过端口 0x3b5 访问的各个数据寄存器(r0-r17)。
0x3d5	写	CRT(6845)数据寄存器。其中数据寄存器 r12-r15 还可以读。 各个数据寄存器的功能说明见下表。
0x3d8	读/写	模式控制寄存器。 位 7-6 未用； 位 5=1 允许闪烁； 位 4=1 640*200 图形模式； 位 3=1 允许视频； 位 2=1 单色显示； 位 1=1 图形模式；=0 文本模式； 位 0=1 80*25 文本模式；=0 40*25 文本模式。
0x3d9	读/写	CGA 调色板寄存器。选择所采用的色彩。 位 7-6 未用； 位 5=1 激活色彩集：青(cyan)、紫(magenta)、白(white)； =0 激活色彩集：红(red)、绿(green)、蓝(blue)； 位 4=1 增强显示图形、文本背景色彩； 位 3=1 增强显示 40*25 的边框、320*200 的背景、640*200 的前景颜色； 位 2=1 显示红色：40*25 的边框、320*200 的背景、640*200 的前景； 位 1=1 显示绿色：40*25 的边框、320*200 的背景、640*200 的前景； 位 0=1 显示蓝色：40*25 的边框、320*200 的背景、640*200 的前景；
0x3da	读	CGA 显示状态寄存器。 位 7-4 未用； 位 3=1 在垂直回扫阶段； 位 2=1 光笔开关关闭；=0 光笔开关接通； 位 1=1 光笔选通有效； 位 0=1 可以不干扰显示访问显示内存；=0 此时不要使用显示内存。
0x3db	写	清除光笔锁存（复位光笔寄存器）。
0x3dc	读/写	预设置光笔锁存（强制光笔选通有效）。

表 7.8 MC6845 内部数据寄存器及初始值

编号	名称	单位	读/写	40*25 模式	80*25 模式	图形模式
r0	水平字符总数	字符	写	0x38	0x71	0x38
r1	水平显示字符数	字符	写	0x28	0x50	0x28
r2	水平同步位置	字符	写	0x2d	0x5a	0x2d
r3	水平同步脉冲宽度	字符	写	0x0a	0x0a	0x0a
r4	垂直字符总数	字符行	写	0x1f	0x1f	0x7f
r5	垂直同步脉冲宽度	扫描行	写	0x06	0x06	0x06

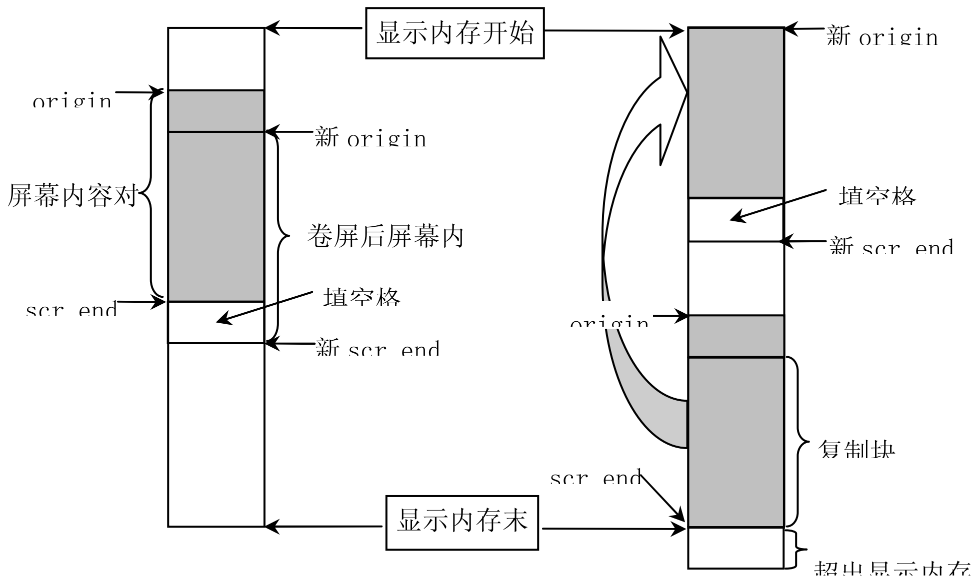
r6 r7	垂直显示字符数 垂直同步位置	字符行 字符行	写 写	0x19 0x1c	0x19 0x1c	0x64 0x70
r8	隔行/逐行选择		写	0x02	0x02	0x02
r9	最大扫描行数	扫描行	写	0x07	0x07	0x01
r10 r11	光标开始位置 光标结束位置	扫描行 扫描行	写 写	0x06 0x07	0x06 0x07	0x06 0x07
r12 r13	显示内存起始位置(高) 显示内存末端位置(低)		写 写	0x00 0x00	0x00 0x00	0x00 0x00
r14 r15	光标当前位置(高) 光标当前位置(低)		读/写 读/写	可变		
r16 r17	光笔当前位置(高) 光笔当前位置(低)		读 读	可变		

7.8.3.2 滚屏操作原理

滚屏操作是指将指定开始行和结束行的一块文本内容向上移动(向上卷动 scroll up)或向下移动(向下卷动 scroll down)，如果将屏幕看作是显示内存上对应屏幕内容的一个窗口的话，那么将屏幕内容向上移即是将窗口沿显示内存向下移动；将屏幕内容向下移动即是将窗口向下移动。在程序中就是重新设置显示控制器中显示内存的起始位置 origin 以及调整程序中相应的变量。对于这两种操作各自都有两种情况。

对于向上卷动，当屏幕对应的显示内存窗口在向下移动后仍然在显示内存范围之内的情况，也即对应当前屏幕的内存块位置始终在显示内存起始位置(video_mem_start)和末端位置 video_mem_end 之间，那么只需要调整显示控制器中起始显示内存位置即可。但是当对应屏幕的内存块位置在向下移动时超出了实际显示内存的末端(video_mem_end)这种情况，就需要移动对应显示内存中的数据，以保证所有当前屏幕数据都落在显示内存范围内。在这第二中情况，程序中是将屏幕对应的内存数据移动到实际显示内存的开始位置处(video_mem_start)。

程序中实际的处理过程分三步进行。首先调整屏幕显示起始位置 origin；然后判断对应屏幕内存数据是否超出显示内存下界(video_mem_end)，如果超出就将屏幕对应的内存数据移动到实际显示内存的开始位置处(video_mem_start)；最后对移动后屏幕上出现的新行用空格字符填满。见下面图 7.4 中所示。其中图(a)对应第一种简单情况，图(b)对应需要移动内存数据时的情况。



(a) 上卷一般情　　　　(b) 需移动屏幕数据的情况

图 7.4 向上卷屏(scroll up)操作示意图

向下卷动屏幕的操作与向上卷屏相似，也会遇到这两种类似情况，只是由于屏幕窗口上移，因此会在屏幕上方出现一空行，并且在屏幕内容所对应的内存超出显示内存范围时需要将屏幕数据内存块往下移动到显示内存的末端位置。

7.8.3.3 ANSI 转义控制序列

终端通常有两部分功能，分别作为计算机信息的输入设备(键盘)和输出设置(显示器)。终端可有许多控制命令，使得终端执行一定的操作而不是仅仅在屏幕上显示一个字符。使用这种方式，计算机就可以命令终端执行移动光标、切换显示模式和响铃等操作。为了能理解程序的执行处理过程，下面对终端控制命令进行一些简单描述。首先说明控制字符和控制序列的含义。

控制字符是指 ASCII 码表开头的 32 个字符(0x00 - 0x1f 或 0-31)以及字符 DEL(0x7f 或 127)，参见附录中的 ASCII 码表。通常一个指定类型的终端都会采用其中的一个子集作为控制字符，而其它的控制字符将不起作用。例如，对于 VT100 终端所采用的控制字符见下表所示。

表 7.9 控制字符

控制字符	八进制	十六进制	Action Taken
NUL	000	0x00	Ignored on input (not stored in input buffer; see full duplex protocol).
ENQ	005	0x05	Transmit answerback message.
BEL	007	0x07	Sound bell tone from keyboard.
BS	010	0x08	Move the cursor to the left one character position, unless it is at the left margin, in which case no action occurs.
HT	011	0x09	Move the cursor to the next tab stop, or to the right margin if no further tab stops are present on the line.
LF	012	0x0a	This code causes a line feed or a new line operation. (See new line mode).
VT	013	0x0b	Interpreted as LF.

FF	014	0x0c	Interpreted as LF.
CR	015	0x0d	Move cursor to the left margin on the current line.
SO	016	0x0e	Invoke G1 character set, as designated by SCS control sequence.
SI	017	0x0f	Select G0 character set, as selected by ESC (sequence.
XON	021	0x11	Causes terminal to resume transmission.
XOFF	023	0x13	Causes terminal to stop transmitted all codes except XOFF and XON.
CAN	030	0x18	If sent during a control sequence, the sequence is immediately terminated and not executed. It also causes the error character to be displayed.
SUB	032	0x1a	Interpreted as CAN.
ESC	033	0x1b	Invokes a control sequence.
DEL	177	0x7f	Ignored on input (not stored in input buffer).

控制序列已经由 ANSI(美国国家标准局 American National Standards Institute)制定为标准：X3.64-1977。控制序列是指由一些非控制字符构成的一个特殊字符序列，终端在收到这个序列时并不是将它们直接显示在屏幕上，而是采取一定的控制操作，比如，移动光标、删除字符、删除行、插入字符或插入行等操作。ANSI 控制序列由以下一些基本元素组成：

控制序列引入码(Control Sequence Introducer - CSI)：表示一个转移序列，提供辅助的控制并且本身是影响随后一系列连续字符含义解释的前缀。通常，一般 CSI 都使用 ESC [。

参数(Parameter)：零个或多个数字字符组成的一个数值。

数值参数(Numeric Parameter)：表示一个数的参数，使用 n 表示。

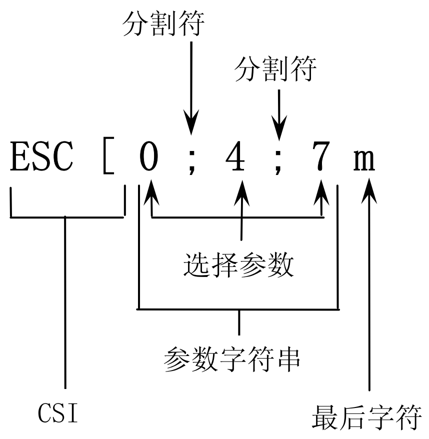
选择参数(Selective Parameter)：用于从一功能子集中选择一个子功能，一般用 s 表示。通常，具有多个选择参数的一个控制序列所产生的作用，如同分立的几个控制序列。例如：CSI sa;sb;sc F 的作用是与 CSI sa F CSI sb F CSI sc F 完全一样的。

参数字符串(Parameter String)：用分号';'隔开的参数字符串。

默认值(Default)：当没有明确指定一个值或者值是 0 的话，就会指定一个与功能相关的值。

最后字符(Final character)：用于结束一个转义或控制序列。

下面是一个控制序列的例子：取消所有字符的属性，然后开启下划线和反显属性。ESC [0;4;7m



下面是常用的一些控制序列列表。

```
Esc Seq Function
======= ==========================================
E[nA    move cursor up n lines
E[nB    move cursor down n lines
E[nC    move cursor right n characters
E[nD    move cursor left n characters
E[n`    move cursor to character position n
E[na    move cursor right n characters
E[nd    move cursor to line n
E[ne    move cursor down n lines
E[nF    move cursor to start of line, n lines up
E[nE    move cursor to start of line, n lines down
E[y;xH  Move cursor to x, y  E[H homes cursor
E[y;xf  Move cursor to x, y
E[nZ    Move cursor back n tab stops
E[nL    Insert n blank lines
E[n@    Insert n blank characters
E[nM    Delete n lines
E[nP    Delete n characters
E[nJ    Erase part or all of display:
        n = 0 from cursor to end of display,
        n = 1 from begin of display to cursor,
        n = 2 entire display.
E[nK    Erase part or all of line:
        n = 0 from cursor to end of line,
        n = 1 from begin of line to cursor,
        n = 2 entire line.
```

```
E[nX    Erase n characters
E[nS    Scroll display n lines up (forward)
E[nT    Scroll display n lines down (reverse)
E[nm    Set character attributes:
        n =  0 normal attribute (all off)
        n =  1 bold
        n =  4 underscore
        n =  5 blink
        n =  7 reverse
        n = 3X set foreground color
        n = 4X set background color
        X = 0 black    X = 1 red
        X = 2 green    X = 3 brown
        X = 4 blue     X = 5 magenta
        X = 6 cyan     X = 7 white
        You can set more than one thing by separating them with a
        semi-colon. eg.  E[0;1;33;40m
E[s     Save cursor position
E[u     Restore saved cursor position
E means 0X1B
if n is 0 then it can also be left off
E[0J  ==  E[J
```

第 8 章 数学协处理器(math)

8.1 概述

列表 8.1 linux/kernel/math 目录

Name	Size	Last modified (GMT)	Description
Makefile	936 bytes	1991-11-18 00:21:45	
math_emulate.c	1023 bytes	1991-11-23 15:36:34	

8.2 Makefile 文件

8.2.1 功能描述

8.2.2 代码注释

列表 8.2 linux/kernel/math/Makefile 文件

```
#
# Makefile for the FREAX-kernel character device drivers.
#
# Note! Dependencies are done automagically by 'make dep', which also
# removes any old dependencies. DON'T put your own dependencies here
# unless it's something special (ie not a .c file).
#
# FREAX(Linux)内核字符设备驱动程序的 Makefile 文件。
# 注意！依赖关系是由'make dep'自动进行的，它也会自动去除原来的依赖信息。不要把你自己的
# 依赖关系信息放在这里，除非是特别文件的（也即不是一个.c 文件的信息）。

AR      =gar     # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。
AS      =gas     # GNU 的汇编程序。
LD      =gld     # GNU 的连接程序。
LDFLAGS =-s -x   # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
CC      =gcc     # GNU C 语言编译器。
  # 下一行是 C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
  # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
  # 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
  # 单短小的函数代码嵌入调用程序中；-mstring-insns Linus 自己填加的优化选项，以后不再使用；
  # -nostdinc -I../include 不使用默认路径中的包含文件，而使用指定目录中的(../../include)。
CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
```

```
15          -finline-functions -mstring-insns -nostdinc -I../../include
    # C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
    # 出设备或指定的输出文件中；-nostdinc -I../../include 同前。
16 CPP      =gcc -E -nostdinc -I../../include
17
    # 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
    # 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
    # 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
    # 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。其中$*.s（或$@）是自动目标变量，
    # $<代表第一个先决条件，这里即是符合条件*.c 的文件。
18 .c.s:
19          $(CC) $(CFLAGS) \
20          -S -o $*.s $<
  # 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。22 行是实现该操作的具体命令。
21 .s.o:
22          $(AS) -c -o $*.o $<
23 .c.o:                       # 类似上面，*.c 文件-→*.o 目标文件。不进行连接。
24          $(CC) $(CFLAGS) \
25          -c -o $*.o $<
26
27 OBJS  = math_emulate.o      # 定义目标文件变量 OBJS。
28
29 math.a: $(OBJS)           # 在有了先决条件 OBJS 后使用下面的命令连接成目标 math.a 库文件。
30          $(AR) rcs math.a $(OBJS)
31          sync
32
    # 下面的规则用于清理工作。当执行'make clean'时，就会执行下面的命令，去除所有编译
    # 连接生成的文件。'rm'是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
33 clean:
34          rm -f core *.o *.a tmp_make
35          for i in *.c;do rm -f `basename $$i .c`.s;done
36
    # 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
    # 使用字符串编辑程序 sed 对 Makefile 文件（即是本文件）进行处理，输出为删除 Makefile
    # 文件中'### Dependencies'行后面的所有行，并生成 tmp_make 临时文件。然后对 kernel/math/
    # 目录下的每个 C 文件执行 gcc 预处理操作.
    # -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
    # 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
    # 文件名加上其依赖关系--该源文件中包含的所有头文件列表。把预处理结果都添加到临时
    # 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。
37 dep:
38          sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
39          (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,'`" "; \
40                  $(CPP) -M $$i;done) >> tmp_make
41          cp tmp_make Makefile
42
43 ### Dependencies:
```

8.3 math-emulation.c 程序

8.3.1 功能描述

8.3.2 代码注释

列表 8.3 linux/kernel/math/math_emulate.c 程序

```
/*
 * linux/kernel/math/math_emulate.c
 *
 * (C) 1991 Linus Torvalds
 */

/*
 * This directory should contain the math-emulation code.
 * Currently only results in a signal.
 */
/*
 * 该目录里应该包含数学仿真代码。目前仅产生一个信号。
 */

#include <signal.h>        // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。

#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h>  // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

//// 协处理器仿真函数。
// 中断处理程序调用的 C 函数，参见(kernel/math/system_call.s，169 行)。
void math_emulate(long edi, long esi, long ebp, long sys_call_ret,
        long eax,long ebx,long ecx,long edx,
        unsigned short fs,unsigned short es,unsigned short ds,
        unsigned long eip,unsigned short cs,unsigned long eflags,
        unsigned short ss, unsigned long esp)
{
        unsigned char first, second;

/* 0x0007 means user code space */
/* 0x0007 表示用户代码空间 */
// 选择符 0x000F 表示在局部描述符表中描述符索引值=1，即代码空间。如果段寄存器 cs 不等于 0x000F
// 则表示 cs 一定是内核代码选择符，是在内核代码空间，则出错，显示此时的 cs:eip 值，并显示信息
// “内核中需要数学仿真”，然后进入死机状态。
        if (cs != 0x000F) {
                printk("math_emulate: %04x:%08x\n\r",cs,eip);
                panic("Math emulation needed in kernel");
        }
// 取用户数据区堆栈数据 first 和 second，显示这些数据，并给进程设置浮点异常信号 SIGFPE。
        first = get_fs_byte((char *)((*&eip)++));
        second = get_fs_byte((char *)((*&eip)++));
```

```
        printk("%04x:%08x %02x %02x\n\r",cs,eip-2,first,second);
        current->signal |= 1<<(SIGFPE-1);
}

//// 协处理器出错处理函数。
// 中断处理程序调用的 C 函数，参见(kernel/math/system_call.s，145 行)。
void math_error(void)
{
// 协处理器指令。(以非等待形式)清除所有异常标志、忙标志和状态字位 7。
        __asm__("fnclex");
// 如果上个任务使用过协处理器，则向上个任务发送协处理器异常信号。
        if (last_task_used_math)
                last_task_used_math->signal |= 1<<(SIGFPE-1);
}

```

第 9 章 文件系统(fs)

9.1 概述

表 9.1 文件系统程序列表

Name	Size	Last modified (GMT)	Description
Makefile	5053 bytes	1991-12-02 03:21:31	m
bitmap.c	4042 bytes	1991-11-26 21:31:53	m
block_dev.c	1422 bytes	1991-10-31 17:19:55	m
buffer.c	9072 bytes	1991-12-06 20:21:00	m
char_dev.c	2103 bytes	1991-11-19 09:10:22	
exec.c	9134 bytes	1991-12-01 20:01:01	m
fcntl.c	1455 bytes	1991-10-02 14:16:29	m
file_dev.c	1852 bytes	1991-12-01 19:02:43	m
file_table.c	122 bytes	1991-10-02 14:16:29	m
inode.c	6933 bytes	1991-12-06 20:16:35	m
ioctl.c	977 bytes	1991-11-19 09:13:05	
namei.c	16562 bytes	1991-11-25 19:19:59	m
open.c	4340 bytes	1991-11-25 19:21:01	m
pipe.c	2385 bytes	1991-10-18 19:02:33	m
read_write.c	2802 bytes	1991-11-25 15:47:20	m
stat.c	1175 bytes	1991-10-02 14:16:29	m
super.c	5628 bytes	1991-12-06 20:10:12	m
truncate.c	1148 bytes	1991-10-02 14:16:29	m

9.2 Makefile 文件

9.2.1 功能描述

9.2.2 代码注释

列表 linux/fs/Makefile 文件

```
AR      =gar     # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。
```

```
2 AS      =gas    # GNU 的汇编程序。
3 CC      =gcc    # GNU C 语言编译器。
4 LD      =gld    # GNU 的连接程序。

  # C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
  # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
  # 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-mstring-insns Linus 自己
  # 填加的优化选项，以后不再使用；-nostdinc -I../include 不使用默认路径中的包含文件，而使
  # 用这里指定目录中的(../include)。
5 CFLAGS  =-Wall -O -fstrength-reduce -fcombine-regs -fomit-frame-pointer \
6         -mstring-insns -nostdinc -I../include

  # C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
  # 出设备或指定的输出文件中；-nostdinc -I../include 同前。
7 CPP     =gcc -E -nostdinc -I../include
8
  # 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
  # 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
  # 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
  # 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。其中$*.s（或$@）是自动目标变量，
  # $<代表第一个先决条件，这里即是符合条件*.c 的文件。
9 .c.s:
10        $(CC) $(CFLAGS) \
11        -S -o $*.s $<
 # 将所有*.c 文件编译成*.o 目标文件。不进行连接。
12 .c.o:
13        $(CC) $(CFLAGS) \
14        -c -o $*.o $<
 # 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。16 行是实现该操作的具体命令。
15 .s.o:
16        $(AS) -o $*.o $<
17
 # 定义目标文件变量 OBJS。
18 OBJS=  open.o read_write.o inode.o file_table.o buffer.o super.o \
19        block_dev.o char_dev.o file_dev.o stat.o exec.o pipe.o namei.o \
20        bitmap.o fcntl.o ioctl.o truncate.o
21
 # 在有了先决条件 OBJS 后使用下面的命令连接成目标 fs.o
22 fs.o: $(OBJS)
23        $(LD) -r -o fs.o $(OBJS)
24
  # 下面的规则用于清理工作。当执行'make clean'时，就会执行 26--27 行上的命令，去除所有编译
  # 连接生成的文件。'rm'是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
25 clean:
26        rm -f core *.o *.a tmp_make
27        for i in *.c;do rm -f `basename $$i .c`.s;done
28
  # 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
  # 使用字符串编辑程序 sed 对 Makefile 文件（这里即是自己）进行处理，输出为删除 Makefile
  # 文件中'### Dependencies'行后面的所有行（下面从 35 开始的行），并生成 tmp_make
  # 临时文件（30 行的作用）。然后对 fs/目录下的每一个 C 文件执行 gcc 预处理操作.
  # -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
  # 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
```

```
# 文件名加上其依赖关系--该源文件中包含的所有头文件列表。把预处理结果都添加到临时
# 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。
dep:
	sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
	(for i in *.c;do $(CPP) -M $$i;done) >> tmp_make
	cp tmp_make Makefile

### Dependencies:
bitmap.o : bitmap.c ../include/string.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
  ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h
block_dev.o : block_dev.c ../include/errno.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
  ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
  ../include/asm/segment.h ../include/asm/system.h
buffer.o : buffer.c ../include/stdarg.h ../include/linux/config.h \
  ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \
  ../include/sys/types.h ../include/linux/mm.h ../include/signal.h \
  ../include/linux/kernel.h ../include/asm/system.h ../include/asm/io.h
char_dev.o : char_dev.c ../include/errno.h ../include/sys/types.h \
  ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \
  ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
  ../include/asm/segment.h ../include/asm/io.h
exec.o : exec.c ../include/errno.h ../include/string.h \
  ../include/sys/stat.h ../include/sys/types.h ../include/a.out.h \
  ../include/linux/fs.h ../include/linux/sched.h ../include/linux/head.h \
  ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
  ../include/asm/segment.h
fcntl.o : fcntl.c ../include/string.h ../include/errno.h \
  ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \
  ../include/sys/types.h ../include/linux/mm.h ../include/signal.h \
  ../include/linux/kernel.h ../include/asm/segment.h ../include/fcntl.h \
  ../include/sys/stat.h
file_dev.o : file_dev.c ../include/errno.h ../include/fcntl.h \
  ../include/sys/types.h ../include/linux/sched.h ../include/linux/head.h \
  ../include/linux/fs.h ../include/linux/mm.h ../include/signal.h \
  ../include/linux/kernel.h ../include/asm/segment.h
file_table.o : file_table.c ../include/linux/fs.h ../include/sys/types.h
inode.o : inode.c ../include/string.h ../include/sys/stat.h \
  ../include/sys/types.h ../include/linux/sched.h ../include/linux/head.h \
  ../include/linux/fs.h ../include/linux/mm.h ../include/signal.h \
  ../include/linux/kernel.h ../include/asm/system.h
ioctl.o : ioctl.c ../include/string.h ../include/errno.h \
  ../include/sys/stat.h ../include/sys/types.h ../include/linux/sched.h \
  ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
  ../include/signal.h
namei.o : namei.c ../include/linux/sched.h ../include/linux/head.h \
  ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
  ../include/signal.h ../include/linux/kernel.h ../include/asm/segment.h \
  ../include/string.h ../include/fcntl.h ../include/errno.h \
  ../include/const.h ../include/sys/stat.h
open.o : open.c ../include/string.h ../include/errno.h ../include/fcntl.h \
  ../include/sys/types.h ../include/utime.h ../include/sys/stat.h \
```

```
	../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \
	../include/linux/mm.h ../include/signal.h ../include/linux/tty.h \
	../include/termios.h ../include/linux/kernel.h ../include/asm/segment.h
pipe.o : pipe.c ../include/signal.h ../include/sys/types.h \
	../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \
	../include/linux/mm.h ../include/asm/segment.h
read_write.o : read_write.c ../include/sys/stat.h ../include/sys/types.h \
	../include/errno.h ../include/linux/kernel.h ../include/linux/sched.h \
	../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
	../include/signal.h ../include/asm/segment.h
stat.o : stat.c ../include/errno.h ../include/sys/stat.h \
	../include/sys/types.h ../include/linux/fs.h ../include/linux/sched.h \
	../include/linux/head.h ../include/linux/mm.h ../include/signal.h \
	../include/linux/kernel.h ../include/asm/segment.h
super.o : super.c ../include/linux/config.h ../include/linux/sched.h \
	../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
	../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
	../include/asm/system.h ../include/errno.h ../include/sys/stat.h
truncate.o : truncate.c ../include/linux/sched.h ../include/linux/head.h \
	../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
	../include/signal.h ../include/sys/stat.h
```

9.2.3 其它信息

9.3 bitmap.c 程序

9.3.1 功能描述

该程序主要用于处理 i 节点和逻辑块（磁盘块或区段）的位图。

9.3.2 代码注释

列表 linux/fs/bitmap.c 程序

```
/*
 *  linux/fs/bitmap.c
 *
 *  (C) 1991  Linus Torvalds
 */

/* bitmap.c contains the code that handles the inode and block bitmaps */
/* bitmap.c 程序含有处理 i 节点和磁盘块位图的代码 */
#include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
                           // 主要使用了其中的 memset()函数。
#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h>  // 内核头文件。含有一些内核常用函数的原形定义。

//// 将指定地址(addr)处的一块内存清零。嵌入汇编程序宏。
// 输入：eax = 0，ecx = 数据块大小 BLOCK_SIZE/4，edi = addr。
```

```
13 #define clear_block(addr) \
14 __asm__("cld\n\t" \              // 清方向位。
15         "rep\n\t" \              // 重复执行存储数据（0）。
16         "stosl" \
17         ::"a" (0),"c" (BLOCK_SIZE/4),"D" ((long) (addr)):"cx","di")
18
   //// 置位指定地址开始的第 nr 个位偏移处的比特位(nr 可以大于 32！)。返回原比特位（0 或 1）。
   // 输入：%0 - eax（返回值），%1 - eax(0)；%2 - nr，位偏移值；%3 - (addr)，addr 的内容。
19 #define set_bit(nr,addr) ({\
20 register int res __asm__("ax"); \
21 __asm__ __volatile__("btsl %2,%3\n\tsetb %%al": \
22 "=a" (res):"" (0),"r" (nr),"m" (*(addr))); \
23 res;})
24
   //// 复位指定地址开始的第 nr 位偏移处的比特位。返回原比特位的反码（1 或 0）。
   // 输入：%0 - eax（返回值），%1 - eax(0)；%2 - nr，位偏移值；%3 - (addr)，addr 的内容。
25 #define clear_bit(nr,addr) ({\
26 register int res __asm__("ax"); \
27 __asm__ __volatile__("btrl %2,%3\n\tsetnb %%al": \
28 "=a" (res):"" (0),"r" (nr),"m" (*(addr))); \
29 res;})
30
   //// 从 addr 开始寻找第 1 个 0 值比特位。
   // 输入：%0 - ecx(返回值)；%1 - ecx(0)；%2 - esi(addr)。
   // 在 addr 指定地址开始的位图中寻找第 1 个是 0 的比特位，并将其距离 addr 的比特位偏移值返回。
31 #define find_first_zero(addr) ({ \
32 int __res; \
33 __asm__("cld\n" \                    // 清方向位。
34         "1:\tlodsl\n\t" \            // 取[esi]➔eax。
35         "notl %%eax\n\t" \           // eax 中每位取反。
36         "bsfl %%eax,%%edx\n\t" \     // 从位 0 扫描 eax 中是 1 的第 1 个位，其偏移值➔edx。
37         "je 2f\n\t" \                // 如果 eax 中全是 0，则向前跳转到标号 2 处(40 行)。
38         "addl %%edx,%%ecx\n\t"\     // 偏移值加入 ecx(ecx 中是位图中首个是 0 的比特位的偏移值)
39         "jmp 3f\n" \                 // 向前跳转到标号 3 处（结束）。
40         "2:\taddl $32,%%ecx\n\t" \   // 没有找到 0 比特位，则将 ecx 加上 1 个长字的位偏移量 32。
41         "cmpl $8192,%%ecx\n\t" \     // 已经扫描了 8192 位（1024 字节）了吗？
42         "jl 1b\n" \                  // 若还没有扫描完 1 块数据，则向前跳转到标号 1 处，继续。
43         "3:" \                       // 结束。此时 ecx 中是位偏移量。
44         :"=c" (__res):"c" (0),"S" (addr):"ax","dx","si"); \
45 __res;})
46
   //// 释放设备 dev 上数据区中的逻辑块 block。
   // 复位指定逻辑块 block 的逻辑块位图比特位。
47 void free_block(int dev, int block)
48 {
49         struct super_block * sb;
50         struct buffer_head * bh;
51
   // 从设备 dev 取超级块，如果指定设备不存在，则出错死机。
52         if (!(sb = get_super(dev)))
53                 panic("trying to free block on nonexistent device");
   // 若逻辑块号小于首个逻辑块号或者大于设备上总逻辑块数，则出错，死机。
54         if (block < sb->s_firstdatazone || block >= sb->s_nzones)
```

```
                panic("trying to free block not in datazone");
// 从 hash 表中寻找该块数据。若找到了则判断其有效性，并清已修改和更新标志，释放该数据块。
        bh = get_hash_table(dev,block);
        if (bh) {
                if (bh->b_count != 1) {
                        printk("trying to free block (%04x:%d), count=%d\n",
                                dev,block,bh->b_count);
                        return;
                }
                bh->b_dirt=0;           // 复位脏（已修改）标志位。
                bh->b_uptodate=0;       // 复位更新标志。
                brelse(bh);
        }
// 计算 block 在数据区开始算起的逻辑块号（从 1 开始计数）。然后对逻辑块(区段)位图进行操作，
// 复位对应的比特位。若对应比特位原来即是 0，则出错，死机。
        block -= sb->s_firstdatazone - 1 ;     // block = block - (    -1) ;
        if (clear_bit(block&8191,sb->s_zmap[block/8192]->b_data)) {
                printk("block (%04x:%d) ",dev,block+sb->s_firstdatazone-1);
                panic("free_block: bit already cleared");
        }
// 置相应逻辑块位图所在缓冲区已修改标志。
        sb->s_zmap[block/8192]->b_dirt = 1;
}

////向设备 dev 申请一个逻辑块（磁盘块，区段）。返回逻辑块号。
 // 置位指定逻辑块 block 的逻辑块位图比特位。
int new_block(int dev)
{
        struct buffer_head * bh;
        struct super_block * sb;
        int i,j;

// 从设备 dev 取超级块，如果指定设备不存在，则出错死机。
        if (!(sb = get_super(dev)))
                panic("trying to get new block from nonexistant device");
// 扫描逻辑块位图，寻找首个 0 比特位，寻找空闲逻辑块，获取放置该逻辑块的块号。。
        j = 8192;
        for (i=0 ; i<8 ; i++)
                if (bh=sb->s_zmap[i])
                        if ((j=find_first_zero(bh->b_data))<8192)
                                break;
// 如果全部扫描完还没找到(i>=8 或 j>=8192)或者位图所在的缓冲块无效(bh=NULL)则 返回 0，
// 退出（没有空闲逻辑块）。
        if (i>=8 || !bh || j>=8192)
                return 0;
// 设置新逻辑块对应逻辑块位图中的比特位，若对应比特位已经置位，则出错，死机。
        if (set_bit(j,bh->b_data))
                panic("new_block: bit already set");
// 置对应缓冲区块的已修改标志。如果新逻辑块大于该设备上的总逻辑块数，则说明指定逻辑块在
// 对应设备上不存在。申请失败，返回 0，退出。
        bh->b_dirt = 1;
        j += i*8192 + sb->s_firstdatazone-1;
        if (j >= sb->s_nzones)
```

```
                return 0;
// 读取设备上的该新逻辑块数据（验证）。如果失败则死机。
        if (!(bh=getblk(dev,j)))
                panic("new_block: cannot get block");
// 新块的引用计数应为 1。否则死机。
        if (bh->b_count != 1)
                panic("new block: count is != 1");
// 将该新逻辑块清零，并置位更新标志和已修改标志。然后释放对应缓冲区，返回逻辑块号。
        clear_block(bh->b_data);
        bh->b_uptodate = 1;
        bh->b_dirt = 1;
        brelse(bh);
        return j;
}

//// 释放指定的 i 节点。
// 复位对应 i 节点位图比特位。
void free_inode(struct m_inode * inode)
{
        struct super_block * sb;
        struct buffer_head * bh;

// 如果 i 节点指针=NULL，则退出。
        if (!inode)
                return;
// 如果 i 节点上的设备号字段为 0，说明该节点无用，则用 0 清空对应 i 节点所占内存区，并返回。
        if (!inode->i_dev) {
                memset(inode,0,sizeof(*inode));
                return;
        }
// 如果此 i 节点还有其它程序引用，则不能释放，说明内核有问题，死机。
        if (inode->i_count>1) {
                printk("trying to free inode with count=%d\n",inode->i_count);
                panic("free_inode");
        }
// 如果文件目录项连接数不为 0，则表示还有其它文件目录项在使用该节点，不应释放，而应该放回等。
        if (inode->i_nlinks)
                panic("trying to free inode with links");
// 取 i 节点所在设备的超级块，测试设备是否存在。
        if (!(sb = get_super(inode->i_dev)))
                panic("trying to free inode on nonexistent device");
// 如果 i 节点号=0 或大于该设备上 i 节点总数，则出错（0 号 i 节点保留没有使用）。
        if (inode->i_num < 1 || inode->i_num > sb->s_ninodes)
                panic("trying to free inode 0 or nonexistant inode");
// 如果该 i 节点对应的节点位图不存在，则出错。
        if (!(bh=sb->s_imap[inode->i_num>>13]))
                panic("nonexistent imap in superblock");
// 复位 i 节点对应的节点位图中的比特位，如果该比特位已经等于 0，则出错。
        if (clear_bit(inode->i_num&8191,bh->b_data))
                printk("free_inode: bit already cleared.\n\r");
// 置 i 节点位图所在缓冲区已修改标志，并清空该 i 节点结构所占内存区。
        bh->b_dirt = 1;
        memset(inode,0,sizeof(*inode));
```

```
}

//// 为设备 dev 建立一个新 i 节点。返回该新 i 节点的指针。
// 在内存 i 节点表中获取一个空闲 i 节点表项，并从 i 节点位图中找一个空闲 i 节点。
struct m_inode * new_inode(int dev)
{
        struct m_inode * inode;
        struct super_block * sb;
        struct buffer_head * bh;
        int i,j;

// 从内存 i 节点表(inode_table)中获取一个空闲 i 节点项(inode)。
        if (!(inode=get_empty_inode()))
                return NULL;
// 读取指定设备的超级块结构。
        if (!(sb = get_super(dev)))
                panic("new_inode with unknown device");
// 扫描 i 节点位图，寻找首个 0 比特位，寻找空闲节点，获取放置该 i 节点的节点号。
        j = 8192;
        for (i=0 ; i<8 ; i++)
                if (bh=sb->s_imap[i])
                        if ((j=find_first_zero(bh->b_data))<8192)
                                break;
// 如果全部扫描完还没找到，或者位图所在的缓冲块无效(bh=NULL)则 返回 0，退出(没有空闲 i 节点)。
        if (!bh || j >= 8192 || j+i*8192 > sb->s_ninodes) {
                iput(inode);
                return NULL;
        }
// 置位对应新 i 节点的 i 节点位图相应比特位，如果已经置位，则出错。
        if (set_bit(j,bh->b_data))
                panic("new_inode: bit already set");
// 置 i 节点位图所在缓冲区已修改标志。
        bh->b_dirt = 1;
// 初始化该 i 节点结构。
        inode->i_count=1;                   // 引用计数。
        inode->i_nlinks=1;                  // 文件目录项链接数。
        inode->i_dev=dev;                   // i 节点所在的设备号。
        inode->i_uid=current->euid;         // i 节点所属用户 id。
        inode->i_gid=current->egid;         // 组 id。
        inode->i_dirt=1;                    // 已修改标志置位。
        inode->i_num = j + i*8192;          // 对应设备中的 i 节点号。
        inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;  // 设置时间。
        return inode;                       // 返回该 i 节点指针。
}

```

9.3.3 其它信息

9.4 inode.c 程序

9.4.1 功能描述

该程序含有处理 i 节点的函数。在注释中，"逻辑块"、"磁盘块"或"区段"的含义相同。

9.4.2 代码注释

列表 linux/fs/inode.c 程序

```
/*
 *  linux/fs/inode.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
#include <sys/stat.h>      // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。

#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h>  // 内核头文件。含有一些内核常用函数的原形定义。
#include <linux/mm.h>      // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
#include <asm/system.h>    // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

struct m_inode inode_table[NR_INODE]={{0,},};     // 内存中 i 节点表（NR_INODE=32 项）。

static void read_inode(struct m_inode * inode);
static void write_inode(struct m_inode * inode);

//// 等待指定的 i 节点可用。
// 如果 i 节点已被锁定，则将当前任务置为不可中断的等待状态。直到该 i 节点解锁。
static inline void wait_on_inode(struct m_inode * inode)
{
        cli();
        while (inode->i_lock)
                sleep_on(&inode->i_wait);
        sti();
}

//// 对指定的 i 节点上锁（锁定指定的 i 节点）。
// 如果 i 节点已被锁定，则将当前任务置为不可中断的等待状态。直到该 i 节点解锁，然后对其上锁。
static inline void lock_inode(struct m_inode * inode)
{
        cli();
        while (inode->i_lock)
                sleep_on(&inode->i_wait);
        inode->i_lock=1;                        // 置锁定标志。
        sti();
}

```

```
//// 对指定的 i 节点解锁。
// 复位 i 节点的锁定标志，并明确地唤醒等待此 i 节点的进程。
static inline void unlock_inode(struct m_inode * inode)
{
        inode->i_lock=0;
        wake_up(&inode->i_wait);
}

//// 释放内存中设备 dev 的所有 i 节点。
// 扫描内存中的 i 节点表数组，如果是指定设备使用的 i 节点就释放之。
void invalidate_inodes(int dev)
{
        int i;
        struct m_inode * inode;

        inode = 0+inode_table;                   // 让指针首先指向 i 节点表指针数组首项。
        for(i=0 ; i<NR_INODE ; i++,inode++) {    // 扫描 i 节点表指针数组中的所有 i 节点。
                wait_on_inode(inode);            // 等待该 i 节点可用（解锁）。
                if (inode->i_dev == dev) {       // 如果是指定设备的 i 节点，则
                        if (inode->i_count)      // 如果其引用数不为 0，则显示出错警告；
                                printk("inode in use on removed disk\n\r");
                        inode->i_dev = inode->i_dirt = 0; // 释放该 i 节点(置设备号为 0 等)。
                }
        }
}

//// 同步所有 i 节点。
// 同步内存与设备上的所有 i 节点信息。
void sync_inodes(void)
{
        int i;
        struct m_inode * inode;

        inode = 0+inode_table;                   // 让指针首先指向 i 节点表指针数组首项。
        for(i=0 ; i<NR_INODE ; i++,inode++) {    // 扫描 i 节点表指针数组。
                wait_on_inode(inode);            // 等待该 i 节点可用（解锁）。
                if (inode->i_dirt && !inode->i_pipe)  // 如果该 i 节点已修改且不是管道节点，
                        write_inode(inode);           // 则写盘。
        }
}

//// 块映射处理操作。(block 位图处理函数，bmap - block map)
// 参数：inode – i 节点指针；block – 数据块号；create - 创建标志。
// 如果创建标志置位，则在对应逻辑块不存在时就申请新磁盘块。
// 返回 block 数据块对应在设备上的逻辑块号。
static int _bmap(struct m_inode * inode,int block,int create)
{
        struct buffer_head * bh;
        int i;

// 如果块号小于 0，则死机。
        if (block<0)
                panic("_bmap: block<0");
```

```
// 如果块号大于直接块数 + 间接块数 + 二次间接块数，超出文件系统表示范围，则死机。
        if (block >= 7+512+512*512)
                panic("_bmap: block>big");

// 如果该块号小于 7，则使用直接块表示。
        if (block<7) {
// 如果创建标志置位，并且 i 节点中对应该块的逻辑块（区段）字段为 0，则向相应设备申请一磁盘
// 块（逻辑块，区段），并将磁盘上的实际逻辑块号填入逻辑块字段中。然后设置 i 节点修改时间，
// 置 i 节点已修改标志。最后返回磁盘上实际逻辑块号。
                if (create && !inode->i_zone[block])
                        if (inode->i_zone[block]=new_block(inode->i_dev)) {
                                inode->i_ctime=CURRENT_TIME;
                                inode->i_dirt=1;
                        }
                return inode->i_zone[block];
        }

// 如果该块号>=7，则如果该块小于 7+512，则说明是一次间接块。下面对一次间接块进行处理。
        block -= 7;
        if (block<512) {
// 如果是创建，并且该 i 节点中对应间接块字段为 0，表明文件是首次使用间接块，则需申请
// 一磁盘块用于存放间接块信息，并将此实际磁盘块号填入间接块字段中。然后设置 i 节点
// 已修改标志和修改时间。
                if (create && !inode->i_zone[7])
                        if (inode->i_zone[7]=new_block(inode->i_dev)) {
                                inode->i_dirt=1;
                                inode->i_ctime=CURRENT_TIME;
                        }
// 若此时 i 节点间接块字段中为 0，表明申请磁盘块失败，返回 0 退出。
                if (!inode->i_zone[7])
                        return 0;
// 读取设备上的一次间接块。
                if (!(bh = bread(inode->i_dev,inode->i_zone[7])))
                        return 0;
// 取该间接块上第 block 项中的逻辑块号。
                i = ((unsigned short *) (bh->b_data))[block];
// 如果是创建并且间接块的第 block 项中的逻辑块号为 0 的话，则申请一磁盘块（逻辑块），并让
// 间接块中的第 block 项等于该新逻辑块块号。然后置位间接块的已修改标志。
                if (create && !i)
                        if (i=new_block(inode->i_dev)) {
                                ((unsigned short *) (bh->b_data))[block]=i;
                                bh->b_dirt=1;
                        }
// 最后释放该间接块，返回磁盘上新申请的对应 block 的逻辑块的块号。
                brelse(bh);
                return i;
        }

// 程序运行到此，表明数据块是二次间接块，处理过程与一次间接块类似。下面是对二次间接块的处理。
// 将 block 再减去间接块所容纳的块数(512)。
        block -= 512;
// 如果是新创建并且 i 节点的二次间接块字段为 0，则需申请一磁盘块用于存放二次间接块的一级块
// 信息，并将此实际磁盘块号填入二次间接块字段中。之后置 i 节点已修改编制和修改时间。
```

```
        if (create && !inode->i_zone[8])
                if (inode->i_zone[8]=new_block(inode->i_dev)) {
                        inode->i_dirt=1;
                        inode->i_ctime=CURRENT_TIME;
                }
// 若此时 i 节点二次间接块字段为 0，表明申请磁盘块失败，返回 0 退出。
        if (!inode->i_zone[8])
                return 0;
// 读取该二次间接块的一级块。
        if (!(bh=bread(inode->i_dev,inode->i_zone[8])))
                return 0;
// 取该二次间接块的一级块上第(block/512)项中的逻辑块号。
        i = ((unsigned short *)bh->b_data)[block>>9];
// 如果是创建并且二次间接块的一级块上第(block/512)项中的逻辑块号为 0 的话，则需申请一磁盘
// 块（逻辑块）作为二次间接块的二级块，并让二次间接块的一级块中第(block/512)项等于该二级
// 块的块号。然后置位二次间接块的一级块已修改标志。并释放二次间接块的一级块。
        if (create && !i)
                if (i=new_block(inode->i_dev)) {
                        ((unsigned short *) (bh->b_data))[block>>9]=i;
                        bh->b_dirt=1;
                }
        brelse(bh);
// 如果二次间接块的二级块块号为 0，表示申请磁盘块失败，返回 0 退出。
        if (!i)
                return 0;
// 读取二次间接块的二级块。
        if (!(bh=bread(inode->i_dev,i)))
                return 0;
// 取该二级块上第 block 项中的逻辑块号。(与上 511 是为了限定 block 值不超过 511)
        i = ((unsigned short *)bh->b_data)[block&511];
// 如果是创建并且二级块的第 block 项中的逻辑块号为 0 的话，则申请一磁盘块（逻辑块），作为
// 最终存放数据信息的块。并让二级块中的第 block 项等于该新逻辑块块号(i)。然后置位二级块的
// 已修改标志。
        if (create && !i)
                if (i=new_block(inode->i_dev)) {
                        ((unsigned short *) (bh->b_data))[block&511]=i;
                        bh->b_dirt=1;
                }
// 最后释放该二次间接块的二级块，返回磁盘上新申请的对应 block 的逻辑块的块号。
        brelse(bh);
        return i;
}

//// 根据 i 节点信息取数据块 block 在设备上对应的逻辑块号。
int bmap(struct m_inode * inode,int block)
{
        return _bmap(inode,block,0);
}

//// 创建数据块 block 在设备上对应的逻辑块，并返回在设备上的逻辑块号。
int create_block(struct m_inode * inode, int block)
{
        return _bmap(inode,block,1);
```

```
}

//// 释放一个 i 节点(回写入设备)。
void iput(struct m_inode * inode)
{
        if (!inode)
                return;
        wait_on_inode(inode);     // 等待 inode 节点解锁(如果已上锁的话)。
        if (!inode->i_count)
                panic("iput: trying to free free inode");
// 如果是管道 i 节点，则唤醒等待该管道的进程，引用次数减 1，如果还有引用则返回。否则释放
// 管道占用的内存页面，并复位该节点的引用计数值、已修改标志和管道标志，并返回。
        if (inode->i_pipe) {
                wake_up(&inode->i_wait);
                if (--inode->i_count)
                        return;
                free_page(inode->i_size);
                inode->i_count=0;
                inode->i_dirt=0;
                inode->i_pipe=0;
                return;
        }
// 如果 i 节点对应的设备号=0，则将此节点的引用计数递减 1，返回。
        if (!inode->i_dev) {
                inode->i_count--;
                return;
        }
// 如果是块设备文件的 i 节点，此时逻辑块字段 0 中是设备号，则刷新该设备。并等待 i 节点解锁。
        if (S_ISBLK(inode->i_mode)) {
                sync_dev(inode->i_zone[0]);
                wait_on_inode(inode);
        }
repeat:
// 如果 i 节点的引用计数大于 1，则递减 1。
        if (inode->i_count>1) {
                inode->i_count--;
                return;
        }
// 如果 i 节点的链接数为 0，则释放该 i 节点的所有逻辑块，并释放该 i 节点。
        if (!inode->i_nlinks) {
                truncate(inode);
                free_inode(inode);
                return;
        }
// 如果该 i 节点已作过修改，则更新该 i 节点，并等待该 i 节点解锁。
        if (inode->i_dirt) {
                write_inode(inode);     /* we can sleep - so do again */
                wait_on_inode(inode);
                goto repeat;
        }
// i 节点引用计数递减 1。
        inode->i_count--;
        return;
```

```
}

//// 从 i 节点表(inode_table)中获取一个空闲 i 节点项。
// 寻找引用计数 count 为 0 的 i 节点，并将其写盘后清零，返回其指针。
struct m_inode * get_empty_inode(void)
{
        struct m_inode * inode;
        static struct m_inode * last_inode = inode_table;  // last_inode 指向 i 节点表第一项。
        int i;

        do {
// 扫描 i 节点表。
                inode = NULL;
                for (i = NR_INODE; i ; i--) {
// 如果 last_inode 已经指向 i 节点表的最后 1 项之后，则让其重新指向 i 节点表开始处。
                        if (++last_inode >= inode_table + NR_INODE)
                                last_inode = inode_table;
// 如果 last_inode 所指向的 i 节点的计数值为 0，则说明可能找到空闲 i 节点项。让 inode 指向
// 该 i 节点。如果该 i 节点的已修改标志和锁定标志均为 0，则我们可以使用该 i 节点，于是退出循环。
                        if (!last_inode->i_count) {
                                inode = last_inode;
                                if (!inode->i_dirt && !inode->i_lock)
                                        break;
                        }
                }
// 如果没有找到空闲 i 节点(inode=NULL)，则将整个 i 节点表打印出来供调试使用，并死机。
                if (!inode) {
                        for (i=0 ; i<NR_INODE ; i++)
                                printk("%04x: %6d\t",inode_table[i].i_dev,
                                        inode_table[i].i_num);
                        panic("No free inodes in mem");
                }
// 等待该 i 节点解锁（如果又被上锁的话）。
                wait_on_inode(inode);
// 如果该 i 节点已修改标志被置位的话，则将该 i 节点刷新，并等待该 i 节点解锁。
                while (inode->i_dirt) {
                        write_inode(inode);
                        wait_on_inode(inode);
                }
        } while (inode->i_count);   // 如果 i 节点又被其它占用的话，则重新寻找空闲 i 节点。
// 已找到空闲 i 节点项。则将该 i 节点项内容清零，并置引用标志为 1，返回该 i 节点指针。
        memset(inode,0,sizeof(*inode));
        inode->i_count = 1;
        return inode;
}

//// 获取管道节点。返回为 i 节点指针（如果是 NULL 则失败）。
// 首先扫描 i 节点表，寻找一个空闲 i 节点项，然后取得一页空闲内存供管道使用。
// 然后将得到的 i 节点的引用计数置为 2(读者和写者)，初始化管道头和尾，置 i 节点的管道类型表示。
struct m_inode * get_pipe_inode(void)
{
        struct m_inode * inode;

```

```
        if (!(inode = get_empty_inode()))         // 如果找不到空闲 i 节点则返回 NULL。
                return NULL;
        if (!(inode->i_size=get_free_page())) {   // 节点的 i_size 字段指向缓冲区。
                inode->i_count = 0;               // 如果已没有空闲内存，则
                return NULL;                      // 释放该 i 节点，并返回 NULL。
        }
        inode->i_count = 2;     /* sum of readers/writers */  /* 读/写两者总计 */
        PIPE_HEAD(*inode) = PIPE_TAIL(*inode) = 0;
        inode->i_pipe = 1;                        // 置节点为管道使用的标志。
        return inode;                             // 返回 i 节点指针。
}

//// 从设备上读取指定节点号的 i 节点。
// nr - i 节点号。
struct m_inode * iget(int dev,int nr)
{
        struct m_inode * inode, * empty;

        if (!dev)
                panic("iget with dev==0");
// 从 i 节点表中取一个空闲 i 节点。
        empty = get_empty_inode();
// 扫描 i 节点表。寻找指定节点号的 i 节点。并递增该节点的引用次数。
        inode = inode_table;
        while (inode < NR_INODE+inode_table) {
// 如果当前扫描的 i 节点的设备号不等于指定的设备号或者节点号不等于指定的节点号，则继续扫描。
                if (inode->i_dev != dev || inode->i_num != nr) {
                        inode++;
                        continue;
                }
// 找到指定设备号和节点号的 i 节点，等待该节点解锁（如果已上锁的话）。
                wait_on_inode(inode);
// 在等待该节点解锁的阶段，节点表可能会发生变化，所以再次判断，如果发生了变化，则再次重新
// 扫描整个 i 节点表。
                if (inode->i_dev != dev || inode->i_num != nr) {
                        inode = inode_table;
                        continue;
                }
// 将该 i 节点引用计数增 1。
                inode->i_count++;
                if (inode->i_mount) {
                        int i;

// 如果该 i 节点是其它文件系统的安装点，则在超级块表中搜寻安装在此 i 节点的超级块。如果没有
// 找到，则显示出错信息，并释放函数开始获取的空闲节点，返回该 i 节点指针。
                        for (i = 0 ; i<NR_SUPER ; i++)
                                if (super_block[i].s_imount==inode)
                                        break;
                        if (i >= NR_SUPER) {
                                printk("Mounted inode hasn't got sb\n");
                                if (empty)
                                        iput(empty);
                                return inode;
```

```
			}
// 将该 i 节点写盘。从安装在此 i 节点文件系统的超级块上取设备号，并令 i 节点号为 1。然后重新
// 扫描整个 i 节点表，取该被安装文件系统的根节点。
			iput(inode);
			dev = super_block[i].s_dev;
			nr = ROOT_INO;
			inode = inode_table;
			continue;
		}
// 已经找到相应的 i 节点，因此放弃临时申请的空闲节点，返回该找到的 i 节点。
		if (empty)
			iput(empty);
		return inode;
	}
// 如果在 i 节点表中没有找到指定的 i 节点，则利用前面申请的空闲 i 节点在 i 节点表中建立该节点。
// 并从相应设备上读取该 i 节点信息。返回该 i 节点。
	if (!empty)
		return (NULL);
	inode=empty;
	inode->i_dev = dev;
	inode->i_num = nr;
	read_inode(inode);
	return inode;
}

//// 从设备上读取指定 i 节点的信息到内存中。
static void read_inode(struct m_inode * inode)
{
	struct super_block * sb;
	struct buffer_head * bh;
	int block;

// 首先锁定该 i 节点，取该节点所在设备的超级块。
	lock_inode(inode);
	if (!(sb=get_super(inode->i_dev)))
		panic("trying to read inode without dev");
// 该 i 节点所在的逻辑块号 = (启动块+超级块) + i 节点位图占用的块数 + 逻辑块位图占用的块数 +
// (i 节点号-1)/每块含有的 i 节点数。
	block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
		(inode->i_num-1)/INODES_PER_BLOCK;
// 从设备上读取该 i 节点所在的逻辑块，并将该 inode 指针指向对应 i 节点信息。
	if (!(bh=bread(inode->i_dev,block)))
		panic("unable to read i-node block");
	*(struct d_inode *)inode =
		((struct d_inode *)bh->b_data)
			[(inode->i_num-1)%INODES_PER_BLOCK];
// 最后释放读入的缓冲区，并解锁该 i 节点。
	brelse(bh);
	unlock_inode(inode);
}

//// 将指定 i 节点信息写入设备（写入对应的缓冲区中，待缓冲区刷新时会写入盘中）。
static void write_inode(struct m_inode * inode)
```

```
{
        struct super_block * sb;
        struct buffer_head * bh;
        int block;

// 首先锁定该 i 节点，如果该 i 节点没有被修改过或者该 i 节点的设备号等于零，则解锁该 i 节点，
// 并退出。
        lock_inode(inode);
        if (!inode->i_dirt || !inode->i_dev) {
                unlock_inode(inode);
                return;
        }
// 获取该 i 节点的超级块。
        if (!(sb=get_super(inode->i_dev)))
                panic("trying to write inode without device");
// 该 i 节点所在的逻辑块号 = (启动块+超级块) + i 节点位图占用的块数 + 逻辑块位图占用的块数 +
// (i 节点号-1)/每块含有的 i 节点数。
        block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
                (inode->i_num-1)/INODES_PER_BLOCK;
// 从设备上读取该 i 节点所在的逻辑块。
        if (!(bh=bread(inode->i_dev,block)))
                panic("unable to read i-node block");
// 将该逻辑块对应该 i 节点的项指向该 i 节点信息。
        ((struct d_inode *)bh->b_data)
                [(inode->i_num-1)%INODES_PER_BLOCK] =
                        *(struct d_inode *)inode;
// 置缓冲区已修改标志，而 i 节点修改标志置零。然后释放该含有 i 节点的缓冲区，并解锁该 i 节点。
        bh->b_dirt=1;
        inode->i_dirt=0;
        brelse(bh);
        unlock_inode(inode);
}

```

9.4.3 其它信息

9.5 buffer.c 文件

9.5.1 功能描述

该文件中的函数主要用于对设备高速缓冲的操作和处理。

空闲缓冲区链表中的缓冲区，并不是都是空闲的。!! 只有当被写盘刷新、解锁且没有其它进程引用时（引用计数=0），才能挪作它用。

9.5.2 代码注释

列表 linux/fs/buffer.c 程序

```
/*
```

```
 *  linux/fs/buffer.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 *  'buffer.c' implements the buffer-cache functions. Race-conditions have
 * been avoided by NEVER letting a interrupt change a buffer (except for the
 * data, of course), but instead letting the caller do it. NOTE! As interrupts
 * can wake up a caller, some cli-sti sequences are needed to check for
 * sleep-on-calls. These should be extremely quick, though (I hope).
 */
/*
 * 'buffer.c'用于实现缓冲区高速缓存功能。通过不让中断过程改变缓冲区，而是让调用者
 * 来执行，避免了竞争条件（当然除改变数据以外）。注意！由于中断可以唤醒一个调用者，
 * 因此就需要开关中断指令（cli-sti）序列来检测等待调用返回。但需要非常地快(希望是这样)。
 */

/*
 * NOTE! There is one discordant note here: checking floppies for
 * disk change. This is where it fits best, I think, as it should
 * invalidate changed floppy-disk-caches.
 */
/*
 * 注意！这里有一个程序应不属于这里：检测软盘是否更换。但我想这里是
 * 放置该程序最好的地方了，因为它需要使已更换软盘缓冲失效。
 */

#include <stdarg.h>       // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了-个
                          // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于
                          // vsprintf、vprintf、vfprintf 函数。

#include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）可选项。
#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
#include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。

extern int end;           // 由连接程序 ld 生成的表明程序末端的变量。[??]
struct buffer_head * start_buffer = (struct buffer_head *) &end;
struct buffer_head * hash_table[NR_HASH];         // NR_HASH = 307 项。
static struct buffer_head * free_list;
static struct task_struct * buffer_wait = NULL;
int NR_BUFFERS = 0;

//// 等待指定缓冲区解锁。
static inline void wait_on_buffer(struct buffer_head * bh)
{
        cli();                          // 关中断。
        while (bh->b_lock)              // 如果已被上锁，则进程进入睡眠，等待其解锁。
                sleep_on(&bh->b_wait);
        sti();                          // 开中断。
```

```
}

//// 系统调用。同步设备和内存高速缓冲中数据。
int sys_sync(void)
{
        int i;
        struct buffer_head * bh;

        sync_inodes();          /* write out inodes into buffers */ /*将 i 节点写入高速缓冲 */
// 扫描所有高速缓冲区，对于已被修改的缓冲块产生写盘请求，将缓冲中数据与设备中同步。
        bh = start_buffer;
        for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
                wait_on_buffer(bh);             // 等待缓冲区解锁（如果已上锁的话）。
                if (bh->b_dirt)
                        ll_rw_block(WRITE,bh);  // 产生写设备块请求。
        }
        return 0;
}

//// 对指定设备进行高速缓冲数据与设备上数据的同步操作。
int sync_dev(int dev)
{
        int i;
        struct buffer_head * bh;

        bh = start_buffer;
        for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
                if (bh->b_dev != dev)
                        continue;
                wait_on_buffer(bh);
                if (bh->b_dev == dev && bh->b_dirt)
                        ll_rw_block(WRITE,bh);
        }
        sync_inodes();                          // 将 i 节点数据写入高速缓冲。
        bh = start_buffer;
        for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
                if (bh->b_dev != dev)
                        continue;
                wait_on_buffer(bh);
                if (bh->b_dev == dev && bh->b_dirt)
                        ll_rw_block(WRITE,bh);
        }
        return 0;
}

//// 使指定设备在高速缓冲区中的数据无效。
// 扫描高速缓冲中的所有缓冲块，对于指定设备的缓冲区，复位其有效(更新)标志和已修改标志。
void inline invalidate_buffers(int dev)
{
        int i;
        struct buffer_head * bh;

```

```
        bh = start_buffer;
        for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
                if (bh->b_dev != dev)            // 如果不是指定设备的缓冲块，则
                        continue;                // 继续扫描下一块。
                wait_on_buffer(bh);              // 等待该缓冲区解锁（如果已被上锁）。
// 由于进程执行过睡眠等待，所以需要再判断一下缓冲区是否是指定设备的。
                if (bh->b_dev == dev)
                        bh->b_uptodate = bh->b_dirt = 0;
        }
}

/*
 * This routine checks whether a floppy has been changed, and
 * invalidates all buffer-cache-entries in that case. This
 * is a relatively slow routine, so we have to try to minimize using
 * it. Thus it is called only upon a 'mount' or 'open'. This
 * is the best way of combining speed and utility, I think.
 * People changing diskettes in the middle of an operation deserve
 * to loose :-)
 *
 * NOTE! Although currently this is only for floppies, the idea is
 * that any additional removable block-device will use this routine,
 * and that mount/open needn't know that floppies/whatever are
 * special.
 */
/*
 * 该子程序检查一个软盘是否已经被更换，如果已经更换就使高速缓冲中与该软驱
 * 对应的所有缓冲区无效。该子程序相对来说较慢，所以我们要尽量少使用它。
 * 所以仅在执行'mount'或'open'时才调用它。我想这是将速度和实用性相结合的
 * 最好方法。若在操作过程当中更换软盘，会导致数据的丢失，这是咎由自取☺。
 *
 * 注意！尽管目前该子程序仅用于软盘，以后任何可移动介质的块设备都将使用该
 * 程序，mount/open 操作是不需要知道是否是软盘或其它什么特殊介质的。
 */
//// 检查磁盘是否更换，如果已更换就使对应高速缓冲区无效。
void check_disk_change(int dev)
{
        int i;

// 是软盘设备吗？如果不是则退出。
        if (MAJOR(dev) != 2)
                return;
// 测试对应软盘是否已更换，如果没有则退出。
        if (!floppy_change(dev & 0x03))
                return;
// 软盘已经更换，所以释放对应设备的 i 节点位图和逻辑块位图所占的高速缓冲区；并使该设备的
// i 节点和数据块信息所占的高速缓冲区无效。
        for (i=0 ; i<NR_SUPER ; i++)
                if (super_block[i].s_dev == dev)
                        put_super(super_block[i].s_dev);
        invalidate_inodes(dev);
        invalidate_buffers(dev);
}
```

```c

// hash 函数和 hash 表项的计算宏定义。
#define _hashfn(dev,block) (((unsigned)(dev^block))%NR_HASH)
#define hash(dev,block) hash_table[_hashfn(dev,block)]

//// 从 hash 队列和空闲缓冲队列中移走指定的缓冲块。
static inline void remove_from_queues(struct buffer_head * bh)
{
/* remove from hash-queue */
/* 从 hash 队列中移除缓冲块 */
        if (bh->b_next)
                bh->b_next->b_prev = bh->b_prev;
        if (bh->b_prev)
                bh->b_prev->b_next = bh->b_next;
// 如果该缓冲区是该队列的头一个块，则让 hash 表的对应项指向本队列中的下一个缓冲区。
        if (hash(bh->b_dev,bh->b_blocknr) == bh)
                hash(bh->b_dev,bh->b_blocknr) = bh->b_next;
/* remove from free list */
/* 从空闲缓冲区表中移除缓冲块 */
        if (!(bh->b_prev_free) || !(bh->b_next_free))
                panic("Free block list corrupted");
        bh->b_prev_free->b_next_free = bh->b_next_free;
        bh->b_next_free->b_prev_free = bh->b_prev_free;
// 如果空闲链表头指向本缓冲区，则让其指向下一缓冲区。
        if (free_list == bh)
                free_list = bh->b_next_free;
}

//// 将指定缓冲区插入空闲链表尾并放入 hash 队列中。
static inline void insert_into_queues(struct buffer_head * bh)
{
/* put at end of free list */
/* 放在空闲链表末尾处 */
        bh->b_next_free = free_list;
        bh->b_prev_free = free_list->b_prev_free;
        free_list->b_prev_free->b_next_free = bh;
        free_list->b_prev_free = bh;
/* put the buffer in new hash-queue if it has a device */
/* 如果该缓冲块对应一个设备，则将其插入新 hash 队列中 */
        bh->b_prev = NULL;
        bh->b_next = NULL;
        if (!bh->b_dev)
                return;
        bh->b_next = hash(bh->b_dev,bh->b_blocknr);
        hash(bh->b_dev,bh->b_blocknr) = bh;
        bh->b_next->b_prev = bh;
}

//// 在高速缓冲中寻找给定设备和指定块的缓冲区块。
// 如果找到则返回缓冲区块的指针，否则返回 NULL。
static struct buffer_head * find_buffer(int dev, int block)
{
        struct buffer_head * tmp;
```

```

        for (tmp = hash(dev,block) ; tmp != NULL ; tmp = tmp->b_next)
                if (tmp->b_dev==dev && tmp->b_blocknr==block)
                        return tmp;
        return NULL;
}

/*
 * Why like this, I hear you say... The reason is race-conditions.
 * As we don't lock buffers (unless we are readint them, that is),
 * something might happen to it while we sleep (ie a read-error
 * will force it bad). This shouldn't really happen currently, but
 * the code is ready.
 */
/*
 * 代码为什么会是这样子的？我听见你问... 原因是竞争条件。由于我们没有对
 * 缓冲区上锁（除非我们正在读取它们中的数据），那么当我们（进程）睡眠时
 * 缓冲区可能会发生一些问题（例如一个读错误将导致该缓冲区出错）。目前
 * 这种情况实际上是不会发生的，但处理的代码已经准备好了。
 */
////
struct buffer_head * get_hash_table(int dev, int block)
{
        struct buffer_head * bh;

        for (;;) {
// 在高速缓冲中寻找给定设备和指定块的缓冲区，如果没有找到则返回 NULL，退出。
                if (!(bh=find_buffer(dev,block)))
                        return NULL;
// 对该缓冲区增加引用计数，并等待该缓冲区解锁（如果已被上锁）。
                bh->b_count++;
                wait_on_buffer(bh);
// 由于经过了睡眠状态，因此有必要再验证该缓冲区块的正确性，并返回缓冲区头指针。
                if (bh->b_dev == dev && bh->b_blocknr == block)
                        return bh;
// 如果该缓冲区所属的设备号或块号在睡眠时发生了改变，则撤消对它的引用计数，重新寻找。
                bh->b_count--;
        }
}

/*
 * Ok, this is getblk, and it isn't very clear, again to hinder
 * race-conditions. Most of the code is seldom used, (ie repeating),
 * so it should be much more efficient than it looks.
 *
 * The algoritm is changed: hopefully better, and an elusive bug removed.
 */
/*
 * OK，下面是 getblk 函数，该函数的逻辑并不是很清晰，同样也是因为要考虑
 * 竞争条件问题。其中大部分代码很少用到，(例如重复操作语句)，因此它应该
 * 比看上去的样子有效得多。
 *
 * 算法以及作了改变：希望能更好，而且一个难以琢磨的错误已经去除。
```

```
 */
// 下面宏定义用于同时判断缓冲区的修改标志和锁定标志，并且定义修改标志的权重要比锁定标志大。
#define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)
//// 取高速缓冲中指定的缓冲区。
// 检查所指定的缓冲区是否已经在高速缓冲中，如果不在，就需要在高速缓冲中建立一个对应的新项。
// 返回相应缓冲区头指针。
struct buffer_head * getblk(int dev,int block)
{
        struct buffer_head * tmp, * bh;

repeat:
// 搜索 hash 表，如果指定块已经在高速缓冲中，则返回对应缓冲区头指针，退出。
        if (bh = get_hash_table(dev,block))
                return bh;
// 扫描空闲数据块链表，寻找空闲缓冲区。
// 首先让 tmp 指向空闲链表的第一个空闲缓冲区头。
        tmp = free_list;
        do {
// 如果该缓冲区正被使用（引用计数不等于 0），则继续扫描下一项。
                if (tmp->b_count)
                        continue;
// 如果缓冲区头指针 bh 为空，或者 tmp 所指缓冲区头的标志(修改、锁定)少于(小于)bh 头的标志，
// 则让 bh 指向该 tmp 缓冲区头。如果该 tmp 缓冲区头表明缓冲区既没有修改也没有锁定标志置位，
// 则说明已为指定设备上的块取得对应的高速缓冲区，则退出循环。
                if (!bh || BADNESS(tmp)<BADNESS(bh)) {
                        bh = tmp;
                        if (!BADNESS(tmp))
                                break;
                }
/* and repeat until we find something good */  /* 重复操作直到找到适合的缓冲区 */
        } while ((tmp = tmp->b_next_free) != free_list);
// 如果所有缓冲区都正被使用（所有缓冲区的头部引用计数都>0），则睡眠，等待有空闲的缓冲区可用。
        if (!bh) {
                sleep_on(&buffer_wait);
                goto repeat;
        }
// 等待该缓冲区解锁（如果已被上锁的话）。
        wait_on_buffer(bh);
// 如果该缓冲区又被其它任务使用的话，只好重复上述过程。
        if (bh->b_count)
                goto repeat;
// 如果该缓冲区已被修改，则将数据写盘，并再次等待缓冲区解锁。如果该缓冲区又被其它任务使用
// 的话，只好再重复上述过程。
        while (bh->b_dirt) {
                sync_dev(bh->b_dev);
                wait_on_buffer(bh);
                if (bh->b_count)
                        goto repeat;
        }
/* NOTE!! While we slept waiting for this block, somebody else might */
/* already have added "this" block to the cache. check it */
/* 注意！！当进程为了等待该缓冲块而睡眠时，其它进程可能已经将该缓冲块 */
 * 加入进高速缓冲中，所以要对此进行检查。*/
```

```
// 在高速缓冲 hash 表中检查指定设备和块的缓冲区是否已经被加入进去。如果是的话，就再次重复
// 上述过程。
        if (find_buffer(dev,block))
                goto repeat;
/* OK, FINALLY we know that this buffer is the only one of it's kind, */
/* and that it's unused (b_count=0), unlocked (b_lock=0), and clean */
/* OK，最终我们知道该缓冲区是指定参数的唯一一块，*/
/* 而且还没有被使用(b_count=0)，未被上锁(b_lock=0)，并且是干净的（未被修改的）*/
// 于是让我们占用此缓冲区。置引用计数为 1，复位修改标志和有效(更新)标志。
        bh->b_count=1;
        bh->b_dirt=0;
        bh->b_uptodate=0;
// 从 hash 队列和空闲块链表中移出该缓冲区头，让该缓冲区用于指定设备和其上的指定块。
        remove_from_queues(bh);
        bh->b_dev=dev;
        bh->b_blocknr=block;
// 然后根据此新的设备号和块号重新插入空闲链表和 hash 队列新位置处。并最终返回缓冲头指针。
        insert_into_queues(bh);
        return bh;
}

//// 释放指定的缓冲区。
// 等待该缓冲区解锁。引用计数递减 1。唤醒等待空闲缓冲区的进程。
void brelse(struct buffer_head * buf)
{
        if (!buf)                  // 如果缓冲头指针无效则返回。
                return;
        wait_on_buffer(buf);
        if (!(buf->b_count--))
                panic("Trying to free free buffer");
        wake_up(&buffer_wait);
}

/*
 * bread() reads a specified block and returns the buffer that contains
 * it. It returns NULL if the block was unreadable.
 */
/*
 * 从设备上读取指定的数据块并返回含有数据的缓冲区。如果指定的块不存在
 * 则返回 NULL。
 */
//// 从指定设备上读取指定的数据块。
struct buffer_head * bread(int dev,int block)
{
        struct buffer_head * bh;

// 在高速缓冲中申请一块缓冲区。如果返回值是 NULL 指针，表示内核出错，死机。
        if (!(bh=getblk(dev,block)))
                panic("bread: getblk returned NULL\n");
// 如果该缓冲区中的数据是有效的（已更新的）可以直接使用，则返回。
        if (bh->b_uptodate)
                return bh;
// 否则调用 ll_rw_block()函数，产生读设备块请求。并等待缓冲区解锁。
```

```
        ll_rw_block(READ,bh);
        wait_on_buffer(bh);
// 如果该缓冲区已更新，则返回缓冲区头指针，退出。
        if (bh->b_uptodate)
                return bh;
// 否则表明读设备操作失败，释放该缓冲区，返回 NULL 指针，退出。
        brelse(bh);
        return NULL;
}

//// 复制内存块。
// 从 from 地址复制一块数据到 to 位置。
#define COPYBLK(from,to) \
__asm__("cld\n\t" \
        "rep\n\t" \
        "movsl\n\t" \
        ::"c" (BLOCK_SIZE/4),"S" (from),"D" (to) \
        :"cx","di","si")

/*
 * bread_page reads four buffers into memory at the desired address. It's
 * a function of its own, as there is some speed to be got by reading them
 * all at the same time, not waiting for one to be read, and then another
 * etc.
 */
/*
 * bread_page 一次读四个缓冲块内容读到内存指定的地址。它是一个完整的函数，
 * 因为同时读取四块可以获得速度上的好处，不用等着读一块，再读一块了。
 */
//// 读设备上一个页面（4 个缓冲区）的内容到内存指定的地址。
void bread_page(unsigned long address,int dev,int b[4])
{
        struct buffer_head * bh[4];
        int i;

// 循环执行 4 次，读一页内容。
        for (i=0 ; i<4 ; i++)
                if (b[i]) {
// 取高速缓冲中指定设备和块号的缓冲区，如果该缓冲区数据无效则产生读设备请求。
                        if (bh[i] = getblk(dev,b[i]))
                                if (!bh[i]->b_uptodate)
                                        ll_rw_block(READ,bh[i]);
                } else
                        bh[i] = NULL;
// 将 4 块缓冲区上的内容顺序复制到指定地址处。
        for (i=0 ; i<4 ; i++,address += BLOCK_SIZE)
                if (bh[i]) {
                        wait_on_buffer(bh[i]);     // 等待缓冲区解锁(如果已被上锁的话)。
                        if (bh[i]->b_uptodate)     // 如果该缓冲区中数据有效的话，则复制。
                                COPYBLK((unsigned long) bh[i]->b_data,address);
                        brelse(bh[i]);             // 释放该缓冲区。
                }
}
```

```

/*
 * Ok, breada can be used as bread, but additionally to mark other
 * blocks for reading as well. End the argument list with a negative
 * number.
 */
/*
 * OK，breada 可以象 bread 一样使用，但会另外预读一些块。该函数参数列表
 * 需要使用一个负数来表明参数列表的结束。
 */
//// 从指定设备读取指定的一些块。
// 成功时返回第 1 块的缓冲区头指针，否则返回 NULL。
struct buffer_head * breada(int dev,int first, ...)
{
        va_list args;
        struct buffer_head * bh, *tmp;

// 取可变参数表中第 1 个参数（块号）。
        va_start(args,first);
// 取高速缓冲中指定设备和块号的缓冲区。如果该缓冲区数据无效，则发出读设备数据块请求。
        if (!(bh=getblk(dev,first)))
                panic("bread: getblk returned NULL\n");
        if (!bh->b_uptodate)
                ll_rw_block(READ,bh);
// 然后顺序取可变参数表中其它预读块号，并作与上面同样处理，但不引用。
        while ((first=va_arg(args,int))>=0) {
                tmp=getblk(dev,first);
                if (tmp) {
                        if (!tmp->b_uptodate)
                                ll_rw_block(READA,bh);
                        tmp->b_count--;
                }
        }
// 可变参数表中所有参数处理完毕。等待第 1 个缓冲区解锁（如果已被上锁）。
        va_end(args);
        wait_on_buffer(bh);
// 如果缓冲区中数据有效，则返回缓冲区头指针，退出。否则释放该缓冲区，返回 NULL，退出。
        if (bh->b_uptodate)
                return bh;
        brelse(bh);
        return (NULL);
}

//// 缓冲区初始化函数。
// 参数 buffer_end 是指定的缓冲区内存的末端。对于系统有 16MB 内存，则缓冲区末端设置为 4MB。
// 对于系统有 8MB 内存，缓冲区末端设置为 2MB。
void buffer_init(long buffer_end)
{
        struct buffer_head * h = start_buffer;
        void * b;
        int i;

// 如果缓冲区高端等于 1Mb，则由于从 640KB-1MB 被显示内存和 BIOS 占用，因此实际可用缓冲区内存
```

```
// 高端应该是 640KB。否则内存高端一定大于 1MB。
        if (buffer_end == 1<<20)
                b = (void *) (640*1024);
        else
                b = (void *) buffer_end;
// 这段代码用于初始化缓冲区，建立空闲缓冲区环链表，并获取系统中缓冲块的数目。
// h 是指向缓冲头结构的指针，而 h+1 是指向内存地址连续的下一个缓冲头地址，也可以说是指向 h
// 缓冲头的末端外。为了保证有足够长度的内存来存储一个缓冲头结构，需要 b 所指向的内存块
// 地址 >= h 缓冲头的末端，也即要>=h+1。
        while ( (b -= BLOCK_SIZE) >= ((void *) (h+1)) ) {
                h->b_dev = 0;                  // 使用该缓冲区的设备号。
                h->b_dirt = 0;                 // 脏标志，也即缓冲区修改标志。
                h->b_count = 0;                // 该缓冲区引用计数。
                h->b_lock = 0;                 // 缓冲区锁定标志。
                h->b_uptodate = 0;             // 缓冲区更新标志（或称数据有效标志）。
                h->b_wait = NULL;              // 指向等待该缓冲区解锁的进程。
                h->b_next = NULL;              // 指向具有相同 hash 值的下一个缓冲头。
                h->b_prev = NULL;              // 指向具有相同 hash 值的前一个缓冲头。
                h->b_data = (char *) b;        // 指向对应缓冲区数据块（1024 字节）。
                h->b_prev_free = h-1;          // 指向链表中前一项。
                h->b_next_free = h+1;          // 指向链表中下一项。
                h++;                              // h 指向下一新缓冲头位置。
                NR_BUFFERS++;                     // 缓冲区块数累加。
                if (b == (void *) 0x100000)       // 如果地址 b 递减到等于 1MB，则跳过 384KB，
                        b = (void *) 0xA0000;     // 让 b 指向地址 0xA0000(640KB)处。
        }
        h--;                                   // 让 h 指向最后一个有效缓冲头。
        free_list = start_buffer;              // 让空闲链表头指向头一个缓冲区头。
        free_list->b_prev_free = h;           // 链表头的 b_prev_free 指向前一项（即最后一项）。
        h->b_next_free = free_list;            // h 的下一项指针指向第一项，形成一个环链。
// 初始化 hash 表（哈希表、散列表），置表中所有的指针为 NULL。
        for (i=0;i<NR_HASH;i++)
                hash_table[i]=NULL;
}
```

9.5.3 其它信息

9.6 block_dev.c 文件

9.6.1 功能描述

该文件包括 block_read()和 block_write()两个函数。这两个函数是供系统调用函数 read()和 write()调用的，其它地方没有引用。

块设备读写操作所使用的函数之间的层次关系为：

- read()，write()

- block_read()，block_write()，file_read()，file_write()，read_pipe()，write()
- bread()或 breada()
- getblk()
- ll_rw_block()

9.6.2 代码注释

列表 linux/fs/block_dev.c 程序

```
/*
 *  linux/fs/block_dev.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <errno.h>         // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。

#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h>  // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
#include <asm/system.h>    // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

//// 数据块写函数 - 向指定设备从给定偏移处写入指定字节数据。
// 参数：dev - 设备号；pos - 设备文件中偏移量指针；buf - 用户地址空间中缓冲区地址；
//       count - 要传送的字节数。
// 对于内核来说，写操作是向高速缓冲区中写入数据，什么时候数据最终写入设备是由高速缓冲管理
// 程序决定并处理的。另外，因为设备是以块为单位进行读写的，因此对于写开始位置不处于块起始
// 处时，需要先将开始字节所在的整个块读出，然后将需要写的数据从写开始处填写满该块，再将完
// 整的一块数据写盘（即交由高速缓冲程序去处理）。
int block_write(int dev, long * pos, char * buf, int count)
{
// 由 pos 地址换算成开始读写块的块序号 block。并求出需读第 1 字节在该块中的偏移位置 offset。
        int block = *pos >> BLOCK_SIZE_BITS;
        int offset = *pos & (BLOCK_SIZE-1);
        int chars;
        int written = 0;
        struct buffer_head * bh;
        register char * p;

// 针对要写入的字节数 count，循环执行以下操作，直到全部写入。
        while (count>0) {
// 计算在该块中可写入的字节数。如果需要写入的字节数填不满一块，则只需写 count 字节。
                chars = BLOCK_SIZE - offset;
                if (chars > count)
                        chars=count;
// 如果正好要写 1 块数据，则直接申请 1 块高速缓冲块，否则需要读入将被修改的数据块，并预读
// 下两块数据，然后将块号递增 1。
                if (chars == BLOCK_SIZE)
                        bh = getblk(dev,block);
                else
                        bh = breada(dev,block,block+1,block+2,-1);
                block++;
```

```
// 如果写出错，则返回已写字节数，如果没有写入任何字节，则返回出错号（负数）。
                if (!bh)
                        return written?written:-EIO;
// p 指向读出数据块中开始写的位置。若最后写入的数据不足一块，则需从块开始填写（修改）所需
// 的字节，因此这里需置 offset 为零。
                p = offset + bh->b_data;
                offset = 0;
// 将文件中偏移指针前移已写字节数。累加已写字节数 chars。传送计数值减去此次已传送字节数。
                *pos += chars;
                written += chars;
                count -= chars;
// 从用户缓冲区复制 chars 字节到 p 指向的高速缓冲区中开始写入的位置。
                while (chars-->0)
                        *(p++) = get_fs_byte(buf++);
// 置该缓冲区块已修改标志，并释放该缓冲区（也即该缓冲区引用计数递减 1）。
                bh->b_dirt = 1;
                brelse(bh);
        }
        return written;                        // 返回已写入的字节数，正常退出。
}

//// 数据块读函数 - 从指定设备和位置读入指定字节数的数据到高速缓冲中。
int block_read(int dev, unsigned long * pos, char * buf, int count)
{
// 由 pos 地址换算成开始读写块的块序号 block。并求出需读第 1 字节在该块中的偏移位置 offset。
        int block = *pos >> BLOCK_SIZE_BITS;
        int offset = *pos & (BLOCK_SIZE-1);
        int chars;
        int read = 0;
        struct buffer_head * bh;
        register char * p;

// 针对要读入的字节数 count，循环执行以下操作，直到全部读入。
        while (count>0) {
// 计算在该块中需读入的字节数。如果需要读入的字节数不满一块，则只需读 count 字节。
                chars = BLOCK_SIZE-offset;
                if (chars > count)
                        chars = count;
// 读入需要的数据块，并预读下两块数据，如果读操作出错，则返回已读字节数，如果没有读入任何
// 字节，则返回出错号。然后将块号递增 1。
                if (!(bh = breada(dev,block,block+1,block+2,-1)))
                        return read?read:-EIO;
                block++;
// p 指向从设备读出数据块中需要读取的开始位置。若最后需要读取的数据不足一块，则需从块开始
// 读取所需的字节，因此这里需将 offset 置零。
                p = offset + bh->b_data;
                offset = 0;
// 将文件中偏移指针前移已读出字节数 chars。累加已读字节数。传送计数值减去此次已传送字节数。
                *pos += chars;
                read += chars;
                count -= chars;
// 从高速缓冲区中 p 指向的开始位置复制 chars 字节数据到用户缓冲区，并释放该高速缓冲区。
                while (chars-->0)
```

```
                        put_fs_byte(*(p++),buf++);
                brelse(bh);
        }
        return read;                    // 返回已读取的字节数，正常退出。
}

```

9.6.3 其它信息

9.7 file_dev.c 文件

9.7.1 功能描述

该文件包括file_read()和file_write()两个函数。这两个函数是供系统调用函数read()和write()调用的，其它地方没有引用。

9.7.2 代码注释

列表 linux/fs/file_dev.c 程序

```
/*
 *  linux/fs/file_dev.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <errno.h>        // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)
#include <fcntl.h>        // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。

#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

#define MIN(a,b) (((a)<(b))?(a):(b))          // 取 a,b 中的最小值。
#define MAX(a,b) (((a)>(b))?(a):(b))          // 取 a,b 中的最大值。

//// 文件读函数 - 根据 i 节点和文件结构，读设备数据。
// 由 i 节点可以知道设备号，由 filp 结构可以知道文件中当前读写指针位置。buf 指定用户态中
// 缓冲区的位置，count 为需要读取的字节数。返回值是实际读取的字节数，或出错号(小于 0)。
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
{
        int left,chars,nr;
        struct buffer_head * bh;

// 若需要读取的字节计数值小于等于零，则返回。
        if ((left=count)<=0)
                return 0;
```

```
// 若还需要读取的字节数不等于 0，就循环执行以下操作，直到全部读出。
        while (left) {
// 根据 i 节点和文件表结构信息，取数据块文件当前读写位置在设备上对应的逻辑块号 nr。若 nr 不
// 为 0，则从 i 节点指定的设备上读取该逻辑块，如果读操作失败则退出循环。若 nr 为 0，表示指定
// 的数据块不存在，置缓冲块指针为 NULL。
                if (nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE)) {
                        if (!(bh=bread(inode->i_dev,nr)))
                                break;
                } else
                        bh = NULL;
// 计算文件读写指针在数据块中的偏移值 nr，则该块中可读字节数为(BLOCK_SIZE-nr)，然后与还需
// 读取的字节数 left 作比较，其中小值即为本次需读的字节数 chars。若(BLOCK_SIZE-nr)大则说明
// 该块是需要读取的最后一块数据，反之则还需要读取一块数据。
                nr = filp->f_pos % BLOCK_SIZE;
                chars = MIN( BLOCK_SIZE-nr , left );
// 调整读写文件指针。指针前移此次将读取的字节数 chars。剩余字节计数相应减去 chars。
                filp->f_pos += chars;
                left -= chars;
// 若从设备上读到了数据，则将 p 指向读出数据块缓冲区中开始读取的位置，并且复制 chars 字节
// 到用户缓冲区 buf 中。否则往用户缓冲区中填入 chars 个 0 值字节。
                if (bh) {
                        char * p = nr + bh->b_data;
                        while (chars-->0)
                                put_fs_byte(*(p++),buf++);
                        brelse(bh);
                } else {
                        while (chars-->0)
                                put_fs_byte(0,buf++);
                }
        }
// 修改该 i 节点的访问时间为当前时间。返回读取的字节数，若读取字节数为 0，则返回出错号。
        inode->i_atime = CURRENT_TIME;
        return (count-left)?(count-left):-ERROR;
}

//// 文件写函数 - 根据 i 节点和文件结构信息，将用户数据写入指定设备。
// 由 i 节点可以知道设备号，由 filp 结构可以知道文件中当前读写指针位置。buf 指定用户态中
// 缓冲区的位置，count 为需要写入的字节数。返回值是实际写入的字节数，或出错号(小于 0)。
int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
{
        off_t pos;
        int block,c;
        struct buffer_head * bh;
        char * p;
        int i=0;

/*
 * ok, append may not work when many processes are writing at the same time
 * but so what. That way leads to madness anyway.
 */
/*
 * ok，当许多进程同时写时，append 操作可能不行，但那又怎样。不管怎样那样做会
 * 导致混乱一团。
```

```
 */
// 如果是要向文件后添加数据，则将文件读写指针移到文件尾部。否则就将在文件读写指针处写入。
        if (filp->f_flags & O_APPEND)
                pos = inode->i_size;
        else
                pos = filp->f_pos;
// 若已写入字节数 i 小于需要写入的字节数 count，则循环执行以下操作。
        while (i<count) {
// 创建数据块号(pos/BLOCK_SIZE)在设备上对应的逻辑块，并返回在设备上的逻辑块号。如果逻辑
// 块号=0，则表示创建失败，退出循环。
                if (!(block = create_block(inode,pos/BLOCK_SIZE)))
                        break;
// 根据该逻辑块号读取设备上的相应数据块，若出错则退出循环。
                if (!(bh=bread(inode->i_dev,block)))
                        break;
// 求出文件读写指针在数据块中的偏移值 c，将 p 指向读出数据块缓冲区中开始读取的位置。置该
// 缓冲区已修改标志。
                c = pos % BLOCK_SIZE;
                p = c + bh->b_data;
                bh->b_dirt = 1;
// 从开始读写位置到块末共可写入 c=(BLOCK_SIZE-c)个字节。若 c 大于剩余还需写入的字节数
// (count-i)，则此次只需再写入 c=(count-i)即可。
                c = BLOCK_SIZE-c;
                if (c > count-i) c = count-i;
// 文件读写指针前移此次需写入的字节数。如果当前文件读写指针位置值超过了文件的大小，则
// 修改 i 节点中文件大小字段，并置 i 节点已修改标志。
                pos += c;
                if (pos > inode->i_size) {
                        inode->i_size = pos;
                        inode->i_dirt = 1;
                }
// 已写入字节计数累加此次写入的字节数 c。从用户缓冲区 buf 中复制 c 个字节到高速缓冲区中 p
// 指向开始的位置处。然后释放该缓冲区。
                i += c;
                while (c-->0)
                        *(p++) = get_fs_byte(buf++);
                brelse(bh);
        }
// 更改文件修改时间为当前时间。
        inode->i_mtime = CURRENT_TIME;
// 如果此次文件操作是在文件尾添加数据，则仍将文件读写指针指向文件尾，并更改 i 节点修改
// 时间为当前时间。
        if (!(filp->f_flags & O_APPEND)) {
                filp->f_pos = pos;
                inode->i_ctime = CURRENT_TIME;
        }
// 返回写入的字节数，若写入字节数为 0，则返回出错号-1。
        return (i?i:-1);
}
```

9.7.3 其它信息

9.8 file_table.c 文件

9.8.1 功能描述

该程序目前是空的，仅定义了文件表数组。

9.8.2 代码注释

列表 linux/fs/file_table.c 程序

```
/*
 *  linux/fs/file_table.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <linux/fs.h>      // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。

struct file file_table[NR_FILE];   // 文件表数组(64 项)。

```

9.8.3 其它信息

9.9 fcntl.c 文件

9.9.1 功能描述

9.9.2 代码注释

列表 linux/fs/fcntl.c 程序

```
/*
 *  linux/fs/fcntl.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
#include <errno.h>         // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
```

```
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

#include <fcntl.h>        // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
#include <sys/stat.h>     // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。

extern int sys_close(int fd);     // 关闭文件系统调用。(fs/open.c, 192)

//// 复制文件句柄(描述符)。
// 参数 fd 是欲复制的文件句柄，arg 指定新文件句柄的最小数值。
// 返回新文件句柄或出错码。
static int dupfd(unsigned int fd, unsigned int arg)
{
// 如果文件句柄值大于一个程序最多打开文件数 NR_OPEN，或者该句柄的文件结构不存在，则出错，
// 返回出错码并退出。
        if (fd >= NR_OPEN || !current->filp[fd])
                return -EBADF;
// 如果指定的新句柄值 arg 大于最多打开文件数，则出错，返回出错码并退出。
        if (arg >= NR_OPEN)
                return -EINVAL;
// 在当前进程的文件结构指针数组中寻找索引号大于等于 arg 但还没有使用的项。
        while (arg < NR_OPEN)
                if (current->filp[arg])
                        arg++;
                else
                        break;
// 如果找到的新句柄值 arg 大于最多打开文件数，则出错，返回出错码并退出。
        if (arg >= NR_OPEN)
                return -EMFILE;
// 在执行时关闭标志位图中复位该句柄位。也即在运行 exec()类函数时不关闭该句柄。
        current->close_on_exec &= ~(1<<arg);
// 令该文件结构指针等于原句柄 fd 的指针，并将文件引用计数增 1。
        (current->filp[arg] = current->filp[fd])->f_count++;
        return arg;             // 返回新的文件句柄。
}

//// 复制文件句柄系统调用函数。
// 复制指定文件句柄 oldfd 为句柄值等于 newfd 的句柄。如果 newfd 已经打开，则首先关闭之。
int sys_dup2(unsigned int oldfd, unsigned int newfd)
{
        sys_close(newfd);              // 若句柄 newfd 已经打开，则首先关闭之。
        return dupfd(oldfd,newfd);     // 复制并返回新句柄。
}

//// 复制文件句柄系统调用函数。
// 复制指定文件句柄 oldfd 为句柄值最小的未用句柄。
int sys_dup(unsigned int fildes)
{
        return dupfd(fildes,0);
}

//// 文件句柄(描述符)操作系统调用函数。
// 参数 fd 是文件句柄，cmd 是操作命令(参见 include/fcntl.h，23-30 行)。
```

```
int sys_fcntl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
        struct file * filp;

// 如果文件句柄值大于一个程序最多打开文件数 NR_OPEN，或者该句柄的文件结构指针为空，则出错，
// 返回出错码并退出。
        if (fd >= NR_OPEN || !(filp = current->filp[fd]))
                return -EBADF;
// 根据不同命令 cmd 进行分别处理。
        switch (cmd) {
                case F_DUPFD:       // 复制文件句柄。
                        return dupfd(fd,arg);
                case F_GETFD:       // 取文件句柄的执行时关闭标志。
                        return (current->close_on_exec>>fd)&1;
                case F_SETFD:       // 设置句柄执行时关闭标志。arg 位 0 置位是设置，否则关闭。
                        if (arg&1)
                                current->close_on_exec |= (1<<fd);
                        else
                                current->close_on_exec &= ~(1<<fd);
                        return 0;
                case F_GETFL:       // 取文件状态标志和访问模式。
                        return filp->f_flags;
                case F_SETFL:       // 设置文件状态和访问模式(根据 arg 设置添加、非阻塞标志)。
                        filp->f_flags &= ~(O_APPEND | O_NONBLOCK);
                        filp->f_flags |= arg & (O_APPEND | O_NONBLOCK);
                        return 0;
                case F_GETLK:   case F_SETLK:   case F_SETLKW:    // 未实现。
                        return -1;
                default:
                        return -1;
        }
}

```

9.9.3 其它信息

9.10 stat.c 文件

9.10.1 功能描述

该程序实现取文件状态信息系统调用函数 stat()和 fstat()，并将信息存放在用户的文件状态结构缓冲区中。stat()是利用文件名取信息，而 fstat()是使用文件句柄(描述符)来取信息。

9.10.2 代码注释

列表 linux/fs/stat.c 程序

```
/*
 *  linux/fs/stat.c
```

```
 *
 *  (C) 1991  Linus Torvalds
 */

#include <errno.h>         // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
#include <sys/stat.h>      // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。

#include <linux/fs.h>      // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h>  // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

//// 复制文件状态信息。
// 参数 inode 是文件对应的 i 节点，statbuf 是 stat 文件状态结构指针，用于存放取得的状态信息。
static void cp_stat(struct m_inode * inode, struct stat * statbuf)
{
        struct stat tmp;
        int i;

// 首先验证(或分配)存放数据的内存空间。
        verify_area(statbuf,sizeof (* statbuf));
// 然后临时复制相应节点上的信息。
        tmp.st_dev = inode->i_dev;          // 文件所在的设备号。
        tmp.st_ino = inode->i_num;          // 文件 i 节点号。
        tmp.st_mode = inode->i_mode;        // 文件属性。
        tmp.st_nlink = inode->i_nlinks;     // 文件的连接数。
        tmp.st_uid = inode->i_uid;          // 文件的用户 id。
        tmp.st_gid = inode->i_gid;          // 文件的组 id。
        tmp.st_rdev = inode->i_zone[0];     // 设备号(如果文件是特殊的字符文件或块文件)。
        tmp.st_size = inode->i_size;        // 文件大小（字节数）（如果文件是常规文件）。
        tmp.st_atime = inode->i_atime;      // 最后访问时间。
        tmp.st_mtime = inode->i_mtime;      // 最后修改时间。
        tmp.st_ctime = inode->i_ctime;      // 最后节点修改时间。
// 最后将这些状态信息复制到用户缓冲区中。
        for (i=0 ; i<sizeof (tmp) ; i++)
                put_fs_byte(((char *) &tmp)[i],&((char *) statbuf)[i]);
}

//// 文件状态系统调用函数 - 根据文件名获取文件状态信息。
// 参数 filename 是指定的文件名，statbuf 是存放状态信息的缓冲区指针。
// 返回 0，若出错则返回出错码。
int sys_stat(char * filename, struct stat * statbuf)
{
        struct m_inode * inode;

// 首先根据文件名找出对应的 i 节点，若出错则返回错误码。
        if (!(inode=namei(filename)))
                return -ENOENT;
// 将 i 节点上的文件状态信息复制到用户缓冲区中，并释放该 i 节点。
        cp_stat(inode,statbuf);
        iput(inode);
        return 0;
```

```
}

//// 文件状态系统调用 - 根据文件句柄获取文件状态信息。
// 参数 fd 是指定文件的句柄(描述符)，statbuf 是存放状态信息的缓冲区指针。
// 返回 0，若出错则返回出错码。
int sys_fstat(unsigned int fd, struct stat * statbuf)
{
        struct file * f;
        struct m_inode * inode;

// 如果文件句柄值大于一个程序最多打开文件数 NR_OPEN，或者该句柄的文件结构指针为空，或者
// 对应文件结构的 i 节点字段为空，则出错，返回出错码并退出。
        if (fd >= NR_OPEN || !(f=current->filp[fd]) || !(inode=f->f_inode))
                return -EBADF;
// 将 i 节点上的文件状态信息复制到用户缓冲区中。
        cp_stat(inode,statbuf);
        return 0;
}

```

9.10.3 其它信息

9.11 pipe.c 文件

9.11.1 功能描述

本程序执行管道文件的读写操作，并实现了管道系统调用 pipe()。

在初始化管道时，管道 i 节点的 i_size 字段中被设置为指向管道缓冲区的指针，管道数据头部指针存放在 i_zone[0]字段中，而管道数据尾部指针存放在 i_zone[1]字段中。对于读管道操作，数据是从管道尾读出，并使管道尾指针前移读取字节数个位置；对于往管道中的写入操作，数据是向管道头部写入，并使管道头指针前移写入字节数个位置。参见下面的管道示意图。

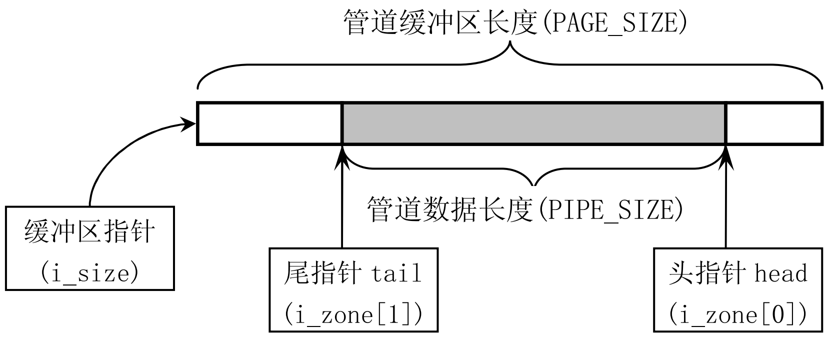


图 5.1 管道缓冲区操作示意图

9.11.2 代码注释

列表 linux/fs/pipe.c 程序

```
/*
 *  linux/fs/pipe.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <signal.h>        // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。

#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/mm.h>    /* for get_free_page */   /* 使用其中的 get_free_page */
                           // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
#include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

//// 管道读操作函数。
// 参数 inode 是管道对应的 i 节点，buf 是数据缓冲区指针，count 是读取的字节数。
int read_pipe(struct m_inode * inode, char * buf, int count)
{
        int chars, size, read = 0;

// 若欲读取的字节计数值 count 大于 0，则循环执行以下操作。
        while (count>0) {
// 若当前管道中没有数据(size=0)，则唤醒等待该节点的进程，如果已没有写管道者，则返回已读
// 字节数，退出。否则在该 i 节点上睡眠，等待信息。
                while (!(size=PIPE_SIZE(*inode))) {
                        wake_up(&inode->i_wait);
                        if (inode->i_count != 2) /* are there any writers? */
                                return read;
                        sleep_on(&inode->i_wait);
                }
// 取管道尾到缓冲区末端的字节数 chars。如果其大于还需要读取的字节数 count，则令其等于 count。
// 如果 chars 大于当前管道中含有数据的长度 size，则令其等于 size。
                chars = PAGE_SIZE-PIPE_TAIL(*inode);
                if (chars > count)
                        chars = count;
                if (chars > size)
                        chars = size;
// 读字节计数减去此次可读的字节数 chars，并累加已读字节数。
                count -= chars;
                read += chars;
// 令 size 指向管道尾部，调整当前管道尾指针（前移 chars 字节）。
                size = PIPE_TAIL(*inode);
                PIPE_TAIL(*inode) += chars;
                PIPE_TAIL(*inode) &= (PAGE_SIZE-1);
// 将管道中的数据复制到用户缓冲区中。对于管道 i 节点，其 i_size 字段中是管道缓冲块指针。
                while (chars-->0)
                        put_fs_byte(((char *)inode->i_size)[size++],buf++);
        }
// 唤醒等待该 i 节点的进程，并返回读取的字节数。
        wake_up(&inode->i_wait);
```

```
        return read;
}

//// 管道写操作函数。
// 参数 inode 是管道对应的 i 节点，buf 是数据缓冲区指针，count 是将写入管道的字节数。
int write_pipe(struct m_inode * inode, char * buf, int count)
{
        int chars, size, written = 0;

// 若将写入的字节计数值 count 还大于 0，则循环执行以下操作。
        while (count>0) {
// 若当前管道中没有已经满了(size=0)，则唤醒等待该节点的进程，如果已没有读管道者，则返回
// 已写入的字节数并退出。若写入 0 字节，则返回-1。否则在该 i 节点上睡眠，等待管道腾出空间。
                while (!(size=(PAGE_SIZE-1)-PIPE_SIZE(*inode))) {
                        wake_up(&inode->i_wait);
                        if (inode->i_count != 2) { /* no readers */
                                current->signal |= (1<<(SIGPIPE-1));
                                return written?written:-1;
                        }
                        sleep_on(&inode->i_wait);
                }
// 取管道头部到缓冲区末端空间字节数 chars。如果其大于还需要写入的字节数 count，则令其等于
// count。如果 chars 大于当前管道中空闲空间长度 size，则令其等于 size。
                chars = PAGE_SIZE-PIPE_HEAD(*inode);
                if (chars > count)
                        chars = count;
                if (chars > size)
                        chars = size;
// 写入字节计数减去此次可写入的字节数 chars，并累加已写字节数到 written。
                count -= chars;
                written += chars;
// 令 size 指向管道数据头部，调整当前管道数据头部指针（前移 chars 字节）。
                size = PIPE_HEAD(*inode);
                PIPE_HEAD(*inode) += chars;
                PIPE_HEAD(*inode) &= (PAGE_SIZE-1);
// 从用户缓冲区复制 chars 个字节到管道中。对于管道 i 节点，其 i_size 字段中是管道缓冲块指针。
                while (chars-->0)
                        ((char *)inode->i_size)[size++]=get_fs_byte(buf++);
        }
// 唤醒等待该 i 节点的进程，返回已写入的字节数，退出。
        wake_up(&inode->i_wait);
        return written;
}

//// 创建管道系统调用函数。
// 在 fildes 所指的数组中创建一对文件句柄(描述符)。这对文件句柄指向一管道 i 节点。fildes[0]
// 用于读管道中数据，fildes[1]用于向管道中写入数据。
// 成功时返回 0，出错时返回-1。
int sys_pipe(unsigned long * fildes)
{
        struct m_inode * inode;
        struct file * f[2];
        int fd[2];
```

```
        int i,j;

// 从系统文件表中取两个空闲项（引用计数字段为 0 的项），并分别设置引用计数为 1。
        j=0;
        for(i=0;j<2 && i<NR_FILE;i++)
                if (!file_table[i].f_count)
                        (f[j++]=i+file_table)->f_count++;
// 如果只有一个空闲项，则释放该项(引用计数复位)。
        if (j==1)
                f[0]->f_count=0;
// 如果没有找到两个空闲项，则返回-1。
        if (j<2)
                return -1;
// 针对上面取得的两个文件结构，分别分配一文件句柄，并使进程的文件结构指针分别指向这两个
// 文件结构。
        j=0;
        for(i=0;j<2 && i<NR_OPEN;i++)
                if (!current->filp[i]) {
                        current->filp[ fd[j]=i ] = f[j];
                        j++;
                }
// 如果只有一个空闲文件句柄，则释放该句柄。
        if (j==1)
                current->filp[fd[0]]=NULL;
// 如果没有找到两个空闲句柄，则释放上面获取的两个文件结构项（复位引用计数值），并返回-1。
        if (j<2) {
                f[0]->f_count=f[1]->f_count=0;
                return -1;
        }
// 申请管道节点。如果不成功，则相应释放两个文件句柄和文件结构项，并返回-1。
        if (!(inode=get_pipe_inode())) {
                current->filp[fd[0]] =
                        current->filp[fd[1]] = NULL;
                f[0]->f_count = f[1]->f_count = 0;
                return -1;
        }
// 初始化两个文件结构，都指向同一个 i 节点，读写指针都置零。第 1 个文件结构的文件模式置为读，
// 第 2 个文件结构的文件模式置为写。
        f[0]->f_inode = f[1]->f_inode = inode;
        f[0]->f_pos = f[1]->f_pos = 0;
        f[0]->f_mode = 1;               /* read */
        f[1]->f_mode = 2;               /* write */
// 将文件句柄数组复制到对应的用户数组中，并返回 0，退出。
        put_fs_long(fd[0],0+fildes);
        put_fs_long(fd[1],1+fildes);
        return 0;
}
```

9.11.3 其它信息

9.12 read_write.c 文件

9.12.1 功能描述

9.12.2 代码注释

列表 linux/fs/read_write.c 程序

```
/*
 *  linux/fs/read_write.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <sys/stat.h>      // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。
#include <errno.h>         // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
#include <sys/types.h>     // 类型头文件。定义了基本的系统数据类型。

#include <linux/kernel.h>  // 内核头文件。含有一些内核常用函数的原形定义。
#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

// 字符设备读写函数。
extern int rw_char(int rw,int dev, char * buf, int count, off_t * pos);
// 读管道操作函数。
extern int read_pipe(struct m_inode * inode, char * buf, int count);
// 写管道操作函数。
extern int write_pipe(struct m_inode * inode, char * buf, int count);
// 块设备读操作函数。
extern int block_read(int dev, off_t * pos, char * buf, int count);
// 块设备写操作函数。
extern int block_write(int dev, off_t * pos, char * buf, int count);
// 读文件操作函数。
extern int file_read(struct m_inode * inode, struct file * filp,
                char * buf, int count);
// 写文件操作函数。
extern int file_write(struct m_inode * inode, struct file * filp,
                char * buf, int count);

//// 重定位文件读写指针系统调用函数。
// 参数 fd 是文件句柄，offset 是新的文件读写指针偏移值，origin 是偏移的起始位置，是 SEEK_SET
// (0，从文件开始处)、SEEK_CUR(1，从当前读写位置)、SEEK_END(2，从文件尾处)三者之一。
int sys_lseek(unsigned int fd,off_t offset, int origin)
{
        struct file * file;
        int tmp;

// 如果文件句柄值大于程序最多打开文件数 NR_OPEN，或者该句柄的文件结构指针为空，或者
// 对应文件结构的 i 节点字段为空，或者指定设备文件指针不可定位，则返回出错码并退出。
        if (fd >= NR_OPEN || !(file=current->filp[fd]) || !(file->f_inode)
```

```
		|| !IS_SEEKABLE(MAJOR(file->f_inode->i_dev)))
			return -EBADF;
// 如果文件对应的 i 节点是管道节点，则返回出错码，退出。管道头尾指针不可随意移动！
	if (file->f_inode->i_pipe)
		return -ESPIPE;
// 根据设置的起始位置，分别重新定位文件读写指针。
	switch (origin) {
// origin = SEEK_SET，要求以文件起始处作为原点设置文件读写指针。若偏移值小于零，则出错返
// 回错误码。否则设置文件读写指针等于 offset。
		case 0:
			if (offset<0) return -EINVAL;
			file->f_pos=offset;
			break;
// origin = SEEK_CUR，要求以文件当前读写指针处作为原点重定位读写指针。如果文件当前指针加
// 上偏移值小于 0，则返回出错码退出。否则在当前读写指针上加上偏移值。
		case 1:
			if (file->f_pos+offset<0) return -EINVAL;
			file->f_pos += offset;
			break;
// origin = SEEK_END，要求以文件末尾作为原点重定位读写指针。此时若文件大小加上偏移值小于零
// 则返回出错码退出。否则重定位读写指针为文件长度加上偏移值。
		case 2:
			if ((tmp=file->f_inode->i_size+offset) < 0)
				return -EINVAL;
			file->f_pos = tmp;
			break;
// origin 设置出错，返回出错码退出。
		default:
			return -EINVAL;
	}
	return file->f_pos;	// 返回重定位后的文件读写指针值。
}

//// 读文件系统调用函数。
// 参数 fd 是文件句柄，buf 是缓冲区，count 是欲读字节数。
int sys_read(unsigned int fd,char * buf,int count)
{
	struct file * file;
	struct m_inode * inode;

// 如果文件句柄值大于程序最多打开文件数 NR_OPEN，或者需要读取的字节计数值小于 0，或者该句柄
// 的文件结构指针为空，则返回出错码并退出。
	if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
		return -EINVAL;
// 若需读取的字节数 count 等于 0，则返回 0，退出
	if (!count)
		return 0;
// 验证存放数据的缓冲区内存限制。
	verify_area(buf,count);
// 取文件对应的 i 节点。若是管道文件，并且是读管道文件模式，则进行读管道操作，若成功则返回
// 读取的字节数，否则返回出错码，退出。
	inode = file->f_inode;
	if (inode->i_pipe)
```

```
                return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
// 如果是字符型文件，则进行读字符设备操作，返回读取的字符数。
        if (S_ISCHR(inode->i_mode))
                return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
// 如果是块设备文件，则执行块设备读操作，并返回读取的字节数。
        if (S_ISBLK(inode->i_mode))
                return block_read(inode->i_zone[0],&file->f_pos,buf,count);
// 如果是目录文件或者是常规文件，则首先验证读取数 count 的有效性并进行调整（若读取字节数加上
// 文件当前读写指针值大于文件大小，则重新设置读取字节数为文件长度-当前读写指针值，若读取数
// 等于 0，则返回 0 退出），然后执行文件读操作，返回读取的字节数并退出。
        if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
                if (count+file->f_pos > inode->i_size)
                        count = inode->i_size - file->f_pos;
                if (count<=0)
                        return 0;
                return file_read(inode,file,buf,count);
        }
// 否则打印节点文件属性，并返回出错码退出。
        printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);
        return -EINVAL;
}

int sys_write(unsigned int fd,char * buf,int count)
{
        struct file * file;
        struct m_inode * inode;

// 如果文件句柄值大于程序最多打开文件数 NR_OPEN，或者需要写入的字节计数小于 0，或者该句柄
// 的文件结构指针为空，则返回出错码并退出。
        if (fd>=NR_OPEN || count <0 || !(file=current->filp[fd]))
                return -EINVAL;
// 若需读取的字节数 count 等于 0，则返回 0，退出
        if (!count)
                return 0;
// 取文件对应的 i 节点。若是管道文件，并且是写管道文件模式，则进行写管道操作，若成功则返回
// 写入的字节数，否则返回出错码，退出。
        inode=file->f_inode;
        if (inode->i_pipe)
                return (file->f_mode&2)?write_pipe(inode,buf,count):-EIO;
// 如果是字符型文件，则进行写字符设备操作，返回写入的字符数，退出。
        if (S_ISCHR(inode->i_mode))
                return rw_char(WRITE,inode->i_zone[0],buf,count,&file->f_pos);
// 如果是块设备文件，则进行块设备写操作，并返回写入的字节数，退出。
        if (S_ISBLK(inode->i_mode))
                return block_write(inode->i_zone[0],&file->f_pos,buf,count);
// 若是常规文件，则执行文件写操作，并返回写入的字节数，退出。
        if (S_ISREG(inode->i_mode))
                return file_write(inode,file,buf,count);
// 否则，显示对应节点的文件模式，返回出错码，退出。
        printk("(Write)inode->i_mode=%06o\n\r",inode->i_mode);
        return -EINVAL;
}
```

9.12.3 其它信息

9.13 truncate.c 文件

9.13.1 功能描述

本程序用于释放指定 i 节点在设备上占用的所有逻辑块，包括直接块、一次间接块和二次间接块。

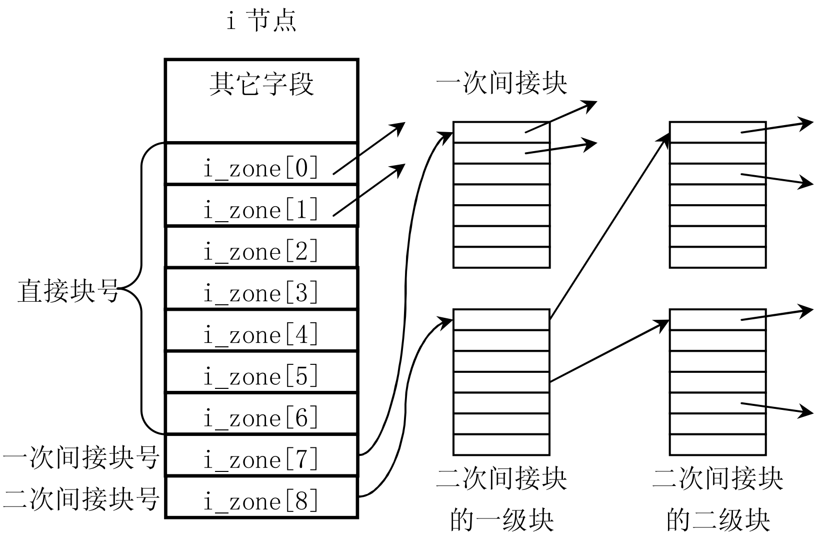


图 索引节点(i 节点)的逻辑块连接方式

9.13.2 代码注释

列表 linux/fs/truncate.c 程序

```
/*
 *  linux/fs/truncate.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。

#include <sys/stat.h>     // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。

//// 释放一次间接块。
static void free_ind(int dev,int block)
{
        struct buffer_head * bh;
        unsigned short * p;
        int i;

// 如果逻辑块号为 0，则返回。
```

```
        if (!block)
                return;
// 读取一次间接块，并释放其上表明使用的所有逻辑块，然后释放该一次间接块的缓冲区。
        if (bh=bread(dev,block)) {
                p = (unsigned short *) bh->b_data;  // 指向数据缓冲区。
                for (i=0;i<512;i++,p++)             // 每个逻辑块上可有 512 个块号。
                        if (*p)
                                free_block(dev,*p); // 释放指定的逻辑块。
                brelse(bh);                         // 释放缓冲区。
        }
// 释放设备上的一次间接块。
        free_block(dev,block);
}

//// 释放二次间接块。
static void free_dind(int dev,int block)
{
        struct buffer_head * bh;
        unsigned short * p;
        int i;

// 如果逻辑块号为 0，则返回。
        if (!block)
                return;
// 读取二次间接块的一级块，并释放其上表明使用的所有逻辑块，然后释放该一级块的缓冲区。
        if (bh=bread(dev,block)) {
                p = (unsigned short *) bh->b_data;  // 指向数据缓冲区。
                for (i=0;i<512;i++,p++)             // 每个逻辑块上可连接 512 个二级块。
                        if (*p)
                                free_ind(dev,*p);   // 释放所有一次间接块。
                brelse(bh);                         // 释放缓冲区。
        }
// 最后释放设备上的二次间接块。
        free_block(dev,block);
}

//// 将节点对应的文件长度截为 0，并释放占用的设备空间。
void truncate(struct m_inode * inode)
{
        int i;

// 如果不是常规文件或者是目录文件，则返回。
        if (!(S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode)))
                return;
// 释放 i 节点的 7 个直接逻辑块，并将这 7 个逻辑块项全置零。
        for (i=0;i<7;i++)
                if (inode->i_zone[i]) {                 // 如果块号不为 0，则释放之。
                        free_block(inode->i_dev,inode->i_zone[i]);
                        inode->i_zone[i]=0;
                }
        free_ind(inode->i_dev,inode->i_zone[7]);        // 释放一次间接块。
        free_dind(inode->i_dev,inode->i_zone[8]);       // 释放二次间接块。
        inode->i_zone[7] = inode->i_zone[8] = 0;        // 逻辑块项 7、8 置零。
```

```
        inode->i_size = 0;                          // 文件大小置零。
        inode->i_dirt = 1;                          // 置节点已修改标志。
        inode->i_mtime = inode->i_ctime = CURRENT_TIME; // 重置文件和节点修改时间为当前时间。
}


```

9.13.3 其它信息

9.14 super.c 文件

9.14.1 功能描述

9.14.2 代码注释

列表 linux/fs/super.c 程序

```
/*
 *  linux/fs/super.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * super.c contains code to handle the super-block tables.
 */
#include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型（HD_TYPE）可选项。
#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

#include <errno.h>        // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
#include <sys/stat.h>     // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。

int sync_dev(int dev);      // 对指定设备执行高速缓冲与设备上数据的同步操作。(fs/buffer.c,59)
void wait_for_keypress(void);  // 等待击键。(kernel/chr_drv/tty_io.c, 140)

/* set_bit uses setb, as gas doesn't recognize setc */
/* set_bit()使用了 setb 指令，因为汇编编译器 gas 不能识别指令 setc */
//// 测试指定位偏移处比特位的值(0 或 1)，并返回该比特位值。(应该取名为 test_bit()更妥帖)
// 嵌入式汇编宏。参数 bitnr 是比特位偏移值，addr 是测试比特位操作的起始地址。
// %0 - ax(__res)，%1 - 0，%2 - bitnr，%3 - addr
#define set_bit(bitnr,addr) ({ \
register int __res __asm__("ax"); \
__asm__("bt %2,%3;setb %%al":"=a" (__res):"a" (0),"r" (bitnr),"m" (*(addr))); \
__res; })

struct super_block super_block[NR_SUPER];    // 超级块结构数组（共 8 项）。
```

```
/* this is initialized in init/main.c */
/* ROOT_DEV 已在 init/main.c 中被初始化 */
int ROOT_DEV = 0;

//// 锁定指定的超级块。
static void lock_super(struct super_block * sb)
{
        cli();                          // 关中断。
        while (sb->s_lock)              // 如果该超级块已经上锁，则睡眠等待。
                sleep_on(&(sb->s_wait));
        sb->s_lock = 1;                 // 给该超级块加锁（置锁定标志）。
        sti();                          // 开中断。
}

//// 对指定超级块解锁。（如果使用 ulock_super 这个名称则更妥帖）。
static void free_super(struct super_block * sb)
{
        cli();                          // 关中断。
        sb->s_lock = 0;                 // 复位锁定标志。
        wake_up(&(sb->s_wait));         // 唤醒等待该超级块的进程。
        sti();                          // 开中断。
}

//// 睡眠等待超级块解锁。
static void wait_on_super(struct super_block * sb)
{
        cli();                          // 关中断。
        while (sb->s_lock)              // 如果超级块已经上锁，则睡眠等待。
                sleep_on(&(sb->s_wait));
        sti();                          // 开中断。
}

//// 取指定设备的超级块。返回该超级块结构指针。
struct super_block * get_super(int dev)
{
        struct super_block * s;

// 如果没有指定设备，则返回空指针。
        if (!dev)
                return NULL;
// s 指向超级块数组开始处。搜索整个超级块数组，寻找指定设备的超级块。
        s = 0+super_block;
        while (s < NR_SUPER+super_block)
// 如果当前搜索项是指定设备的超级块，则首先等待该超级块解锁（若已经被其它进程上锁的话）。
// 在等待期间，该超级块有可能被其它设备使用，因此此时需再判断一次是否是指定设备的超级块，
// 如果是则返回该超级块的指针。否则就重新对超级块数组再搜索一遍，因此 s 重又指向超级块数组
// 开始处。
                if (s->s_dev == dev) {
                        wait_on_super(s);
                        if (s->s_dev == dev)
                                return s;
                        s = 0+super_block;
// 如果当前搜索项不是，则检查下一项。如果没有找到指定的超级块，则返回空指针。
```

```
                } else
                        s++;
        return NULL;
}

//// 释放指定设备的超级块。
// 释放设备所使用的超级块数组项（置 s_dev=0），并释放该设备 i 节点位图和逻辑块位图所占用
// 的高速缓冲块。如果超级块对应的文件系统是根文件系统，或者其 i 节点上已经安装有其它的文件
// 系统，则不能释放该超级块。
void put_super(int dev)
{
        struct super_block * sb;
        struct m_inode * inode;
        int i;

// 如果指定设备是根文件系统设备，则显示警告信息“根系统盘改变了，准备生死决战吧”，并返回。
        if (dev == ROOT_DEV) {
                printk("root diskette changed: prepare for armageddon\n\r");
                return;
        }
// 如果找不到指定设备的超级块，则返回。
        if (!(sb = get_super(dev)))
                return;
// 如果该超级块指明本文件系统 i 节点上安装有其它的文件系统，则显示警告信息，返回。
        if (sb->s_imount) {
                printk("Mounted disk changed - tssk, tssk\n\r");
                return;
        }
// 找到指定设备的超级块后，首先锁定该超级块，然后置该超级块对应的设备号字段为 0，也即即将
// 放弃该超级块。
        lock_super(sb);
        sb->s_dev = 0;
// 然后释放该设备 i 节点位图和逻辑块位图所占用的缓冲块。
        for(i=0;i<I_MAP_SLOTS;i++)
                brelse(sb->s_imap[i]);
        for(i=0;i<Z_MAP_SLOTS;i++)
                brelse(sb->s_zmap[i]);
// 最后对该超级块解锁，并返回。
        free_super(sb);
        return;
}

//// 从设备上读取超级块到内存中。
// 如果该设备的超级块已经在高速缓冲中并且有效，则直接返回该超级块的指针。
static struct super_block * read_super(int dev)
{
        struct super_block * s;
        struct buffer_head * bh;
        int i,block;

// 如果没有指明设备，则返回空指针。
        if (!dev)
                return NULL;
```

```
// 首先检查该设备是否可更换过盘片（也即是否是软盘设备），如果更换过盘，则高速缓冲区有关该
// 设备的所有缓冲块均失效，需要进行失效处理。
        check_disk_change(dev);
// 如果该设备的超级块已经在高速缓冲中，则直接返回该超级块的指针。
        if (s = get_super(dev))
                return s;
// 否则，首先在超级块数组中找出一个空项(也即其 s_dev=0 的项)。如果数组已经占满则返回空指针。
        for (s = 0+super_block ;; s++) {
                if (s >= NR_SUPER+super_block)
                        return NULL;
                if (!s->s_dev)
                        break;
        }
// 找到超级块空项后，就将该超级块用于指定设备，对该超级块进行部分初始化。
        s->s_dev = dev;
        s->s_isup = NULL;
        s->s_imount = NULL;
        s->s_time = 0;
        s->s_rd_only = 0;
        s->s_dirt = 0;
// 然后所定该超级块，并从设备上读取超级块信息到 bh 指向的缓冲区中。如果读超级块操作失败，
// 则释放上面选定的超级块数组中的项，并解锁该项，返回空指针退出。
        lock_super(s);
        if (!(bh = bread(dev,1))) {
                s->s_dev=0;
                free_super(s);
                return NULL;
        }
// 将设备上读取的超级块信息复制到内存超级块结构中。并释放存放读取信息的高速缓冲块。
        *((struct d_super_block *) s) =
                *((struct d_super_block *) bh->b_data);
        brelse(bh);
// 如果读取的超级块的文件系统魔数字段内容不对，说明设备上不是正确的文件系统，因此同上面
// 一样，释放上面选定的超级块数组中的项，并解锁该项，返回空指针退出。
// 对于该版 linux 内核，只支持 minix 文件系统版本 1，其魔数是 0x137f。
        if (s->s_magic != SUPER_MAGIC) {
                s->s_dev = 0;
                free_super(s);
                return NULL;
        }
// 下面开始读取设备上 i 节点位图和逻辑块位图数据。首先初始化内存超级块结构中位图空间。
        for (i=0;i<I_MAP_SLOTS;i++)
                s->s_imap[i] = NULL;
        for (i=0;i<Z_MAP_SLOTS;i++)
                s->s_zmap[i] = NULL;
// 然后从设备上读取 i 节点位图和逻辑块位图信息，并存放在超级块对应字段中。
        block=2;
        for (i=0 ; i < s->s_imap_blocks ; i++)
                if (s->s_imap[i]=bread(dev,block))
                        block++;
                else
                        break;
        for (i=0 ; i < s->s_zmap_blocks ; i++)
```

```
                        if (s->s_zmap[i]=bread(dev,block))
                                block++;
                        else
                                break;
// 如果读出的位图逻辑块数不等于位图应该占有的逻辑块数，说明文件系统位图信息有问题，超级块
// 初始化失败。因此只能释放前面申请的所有资源，返回空指针并退出。
        if (block != 2+s->s_imap_blocks+s->s_zmap_blocks) {
// 释放 i 节点位图和逻辑块位图占用的高速缓冲区。
                for(i=0;i<I_MAP_SLOTS;i++)
                        brelse(s->s_imap[i]);
                for(i=0;i<Z_MAP_SLOTS;i++)
                        brelse(s->s_zmap[i]);
//释放上面选定的超级块数组中的项，并解锁该超级块项，返回空指针退出。
                s->s_dev=0;
                free_super(s);
                return NULL;
        }
// 否则一切成功。对于申请空闲 i 节点的函数来讲，如果设备上所有的 i 节点已经全被使用，则查找
// 函数会返回 0 值。因此 0 号 i 节点是不能用的，所以这里将位图中的最低位设置为 1，以防止文件
// 系统分配 0 号 i 节点。同样的道理，也将逻辑块位图的最低位设置为 1。
        s->s_imap[0]->b_data[0] |= 1;
        s->s_zmap[0]->b_data[0] |= 1;
// 解锁该超级块，并返回超级块指针。
        free_super(s);
        return s;
}

//// 卸载文件系统调用函数。
// 参数 dev_name 是设备文件名。
int sys_umount(char * dev_name)
{
        struct m_inode * inode;
        struct super_block * sb;
        int dev;

// 首先根据设备文件名找到对应的 i 节点，并取其中的设备号。
        if (!(inode=namei(dev_name)))
                return -ENOENT;
        dev = inode->i_zone[0];
// 如果不是块设备文件，则释放刚申请的 i 节点 dev_i，返回出错码。
        if (!S_ISBLK(inode->i_mode)) {
                iput(inode);
                return -ENOTBLK;
        }
// 释放刚申请的 i 节点 dev_i。
        iput(inode);
// 如果设备是根文件系统，则不能被卸载，返回出错号。
        if (dev==ROOT_DEV)
                return -EBUSY;
// 如果取设备的超级块失败，或者该设备文件系统没有安装过，则返回出错码。
        if (!(sb=get_super(dev)) || !(sb->s_imount))
                return -ENOENT;
// 如果超级块所指明的被安装到的 i 节点没有置位其安装标志，则显示警告信息。
```

```
        if (!sb->s_imount->i_mount)
                printk("Mounted inode has i_mount=0\n");
// 查找 i 节点表，看是否有进程在使用该设备上的文件，如果有则返回忙出错码。
        for (inode=inode_table+0 ; inode<inode_table+NR_INODE ; inode++)
                if (inode->i_dev==dev && inode->i_count)
                                return -EBUSY;
// 复位被安装到的 i 节点的安装标志，释放该 i 节点。
        sb->s_imount->i_mount=0;
        iput(sb->s_imount);
// 置超级块中被安装 i 节点字段为空，并释放设备文件系统的根 i 节点，置超级块中被安装系统
// 根 i 节点指针为空。
        sb->s_imount = NULL;
        iput(sb->s_isup);
        sb->s_isup = NULL;
// 释放该设备的超级块，并对该设备执行高速缓冲与设备上数据的同步操作。
        put_super(dev);
        sync_dev(dev);
        return 0;
}

//// 安装文件系统调用函数。
// 参数 dev_name 是设备文件名，dir_name 是安装到的目录名，rw_flag 被安装文件的读写标志。
int sys_mount(char * dev_name, char * dir_name, int rw_flag)
{
        struct m_inode * dev_i, * dir_i;
        struct super_block * sb;
        int dev;

// 首先根据设备文件名找到对应的 i 节点，并取其中的设备号。
        if (!(dev_i=namei(dev_name)))
                return -ENOENT;
        dev = dev_i->i_zone[0];
// 如果不是块设备文件，则释放刚申请的 i 节点 dev_i，返回出错码。
        if (!S_ISBLK(dev_i->i_mode)) {
                iput(dev_i);
                return -EPERM;
        }
// 释放刚申请的 i 节点 dev_i。
        iput(dev_i);
// 根据给定的目录文件名找到对应的 i 节点 dir_i。
        if (!(dir_i=namei(dir_name)))
                return -ENOENT;
// 如果该 i 节点的引用计数不为 1（仅在这里引用），或者该 i 节点的节点号是根文件系统的节点
// 号 1，则释放该 i 节点，返回出错码。
        if (dir_i->i_count != 1 || dir_i->i_num == ROOT_INO) {
                iput(dir_i);
                return -EBUSY;
        }
// 如果该节点不是一个目录文件节点，则也释放该 i 节点，返回出错码。
        if (!S_ISDIR(dir_i->i_mode)) {
                iput(dir_i);
                return -EPERM;
        }
```

```
// 读取将安装文件系统的超级块，如果失败则也释放该 i 节点，返回出错码。
        if (!(sb=read_super(dev))) {
                iput(dir_i);
                return -EBUSY;
        }
// 如果将要被安装的文件系统已经安装在其它地方，则释放该 i 节点，返回出错码。
        if (sb->s_imount) {
                iput(dir_i);
                return -EBUSY;
        }
// 如果将要安装到的 i 节点已经安装了文件系统(安装标志已经置位)，则释放该 i 节点，返回出错码。
        if (dir_i->i_mount) {
                iput(dir_i);
                return -EPERM;
        }
// 被安装文件系统超级块的被安装到 i 节点字段指向安装到的目录的 i 节点。
        sb->s_imount=dir_i;
// 置安装 i 节点的安装标志和节点已修改标志。/* 注意! 这里没有 iput(dir_i) */
        dir_i->i_mount=1;                    /* 这将在 umount 内操作 */
        dir_i->i_dirt=1;                /* NOTE! we don't iput(dir_i) */
        return 0;                       /* we do that in umount */
}

//// 安装根文件系统。
// 该函数是在系统开机初始化设置时(sys_setup())调用的。( kernel/blk_drv/hd.c, 157 )
void mount_root(void)
{
        int i,free;
        struct super_block * p;
        struct m_inode * mi;

// 如果磁盘 i 节点结构不是 32 个字节，则出错，死机。该判断是用于防止修改源代码时的不一致性。
        if (32 != sizeof (struct d_inode))
                panic("bad i-node size");
// 初始化文件表数组（共 64 项，也即系统同时只能打开 64 个文件），将所有文件结构中的引用计数
// 设置为 0。[??为什么放在这里初始化？]
        for(i=0;i<NR_FILE;i++)
                file_table[i].f_count=0;
// 如果根文件系统所在设备是软盘的话，就提示“插入根文件系统盘，并按回车键”，并等待按键。
        if (MAJOR(ROOT_DEV) == 2) {
                printk("Insert root floppy and press ENTER");
                wait_for_keypress();
        }
// 初始化超级块数组（共 8 项）。
        for(p = &super_block[0] ; p < &super_block[NR_SUPER] ; p++) {
                p->s_dev = 0;
                p->s_lock = 0;
                p->s_wait = NULL;
        }
// 如果读设备上超级块失败，则显示信息，并死机。
        if (!(p=read_super(ROOT_DEV)))
                panic("Unable to mount root");
//从设备上读取文件系统的根 i 节点(1)，如果失败则显示出错信息，死机。
```

```
        if (!(mi=iget(ROOT_DEV,ROOT_INO)))
                panic("Unable to read root i-node");
// 该 i 节点引用次数递增 3 次。[?? Why?]
        mi->i_count += 3 ;      /* NOTE! it is logically used 4 times, not 1 */
// 置该超级块的被安装文件系统 i 节点和被安装到的 i 节点为该 i 节点。
        p->s_isup = p->s_imount = mi;
// 设置当前进程的当前工作目录和根目录 i 节点。
        current->pwd = mi;
        current->root = mi;
// 统计该设备上空闲块数。首先令 i 等于超级块中表明的设备逻辑块总数。
        free=0;
        i=p->s_nzones;
// 然后根据逻辑块位图中相应比特位的占用情况统计出空闲块数。宏函数 set_bit()其实只是测试
// 比特位，而非设置比特位。"i&8191"用于取得 i 节点号在当前块中的偏移值。"i>>13"是将 i 除以
// 8192，也即除一个磁盘块包含的比特位数。
        while (-- i >= 0)
                if (!set_bit(i&8191,p->s_zmap[i>>13]->b_data))
                        free++;
// 显示设备上空闲逻辑块数/逻辑块总数。
        printk("%d/%d free blocks\n\r",free,p->s_nzones);
// 统计设备上空闲 i 节点数。首先令 i 等于超级块中表明的设备上 i 节点总数+1。加 1 是将 0 节点
// 也统计进去。[??]
        free=0;
        i=p->s_ninodes+1;
// 然后根据 i 节点位图中相应比特位的占用情况计算出空闲 i 节点数。
        while (-- i >= 0)
                if (!set_bit(i&8191,p->s_imap[i>>13]->b_data))
                        free++;
// 显示设备上可用的空闲 i 节点数/i 节点总数。
        printk("%d/%d free inodes\n\r",free,p->s_ninodes);
}

```

9.14.3 其它信息

9.15 open.c 文件

9.15.1 功能描述

9.15.2 代码注释

列表 linux/fs/open.c 程序

```
/*
 *  linux/fs/open.c
 *
 *  (C) 1991  Linus Torvalds
 */

```

```
#include <string.h>       // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
#include <errno.h>        // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
#include <fcntl.h>        // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
#include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
#include <utime.h>        // 用户时间头文件。定义了访问和修改时间结构以及 utime()原型。
#include <sys/stat.h>     // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。

#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/tty.h>    // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

// 取文件系统信息系统调用函数。
int sys_ustat(int dev, struct ustat * ubuf)
{
        return -ENOSYS;
}

//// 设置文件访问和修改时间。
// 参数 filename 是文件名，times 是访问和修改时间结构指针。
// 如果 times 指针不为 NULL，则取 utimbuf 结构中的时间信息来设置文件的访问和修改时间。如果
// times 指针是 NULL，则取系统当前时间来设置指定文件的访问和修改时间域。
int sys_utime(char * filename, struct utimbuf * times)
{
        struct m_inode * inode;
        long actime,modtime;

// 根据文件名寻找对应的 i 节点，如果没有找到，则返回出错码。
        if (!(inode=namei(filename)))
                return -ENOENT;
// 如果访问和修改时间数据结构指针不为 NULL，则从结构中读取用户设置的时间值。
        if (times) {
                actime = get_fs_long((unsigned long *) &times->actime);
                modtime = get_fs_long((unsigned long *) &times->modtime);
// 否则将访问和修改时间置为当前时间。
        } else
                actime = modtime = CURRENT_TIME;
// 修改 i 节点中的访问时间字段和修改时间字段。
        inode->i_atime = actime;
        inode->i_mtime = modtime;
// 置 i 节点已修改标志，释放该节点，并返回 0。
        inode->i_dirt = 1;
        iput(inode);
        return 0;
}

/*
 * XXX should we use the real or effective uid?  BSD uses the real uid,
 * so as to make this call useful to setuid programs.
 */
/*
 * 文件属性 XXX，我们该用真实用户 id 还是有效用户 id？BSD 系统使用了真实用户 id，
```

```
 * 以使该调用可以供 setuid 程序使用。（注：POSIX 标准建议使用真实用户 ID）
 */
//// 检查对文件的访问权限。
// 参数 filename 是文件名，mode 是屏蔽码，由 R_OK(4)、W_OK(2)、X_OK(1)和 F_OK(0)组成。
// 如果请求访问允许的话，则返回 0，否则返回出错码。
int sys_access(const char * filename,int mode)
{
        struct m_inode * inode;
        int res, i_mode;

// 屏蔽码由低 3 位组成，因此清除所有高比特位。
        mode &= 0007;
// 如果文件名对应的 i 节点不存在，则返回出错码。
        if (!(inode=namei(filename)))
                return -EACCES;
// 取文件的属性码，并释放该 i 节点。
        i_mode = res = inode->i_mode & 0777;
        iput(inode);
// 如果当前进程是该文件的宿主，则取文件宿主属性。
        if (current->uid == inode->i_uid)
                res >>= 6;
// 否则如果当前进程是与该文件同属一组，则取文件组属性。
        else if (current->gid == inode->i_gid)
                res >>= 6;
// 如果文件属性具有查询的属性位，则访问许可，返回 0。
        if ((res & 0007 & mode) == mode)
                return 0;
        /*
         * XXX we are doing this test last because we really should be
         * swapping the effective with the real user id (temporarily),
         * and then calling suser() routine.  If we do call the
         * suser() routine, it needs to be called last.
         */
        /*
         * XXX 我们最后才做下面的测试，因为我们实际上需要交换有效用户 id 和
         * 真实用户 id（临时地），然后才调用 suser()函数。如果我们确实要调用
         * suser()函数，则需要最后才被调用。
         */
// 如果当前用户 id 为 0（超级用户）并且屏蔽码执行位是 0 或文件可以被任何人访问，则返回 0。
        if ((!current->uid) &&
            (!(mode & 1) || (i_mode & 0111)))
                return 0;
// 否则返回出错码。
        return -EACCES;
}

//// 改变当前工作目录系统调用函数。
// 参数 filename 是目录名。
// 操作成功则返回 0，否则返回出错码。
int sys_chdir(const char * filename)
{
        struct m_inode * inode;

```

```
// 如果文件名对应的 i 节点不存在，则返回出错码。
        if (!(inode = namei(filename)))
                return -ENOENT;
// 如果该 i 节点不是目录的 i 节点，则释放该节点，返回出错码。
        if (!S_ISDIR(inode->i_mode)) {
                iput(inode);
                return -ENOTDIR;
        }
// 释放当前进程原工作目录 i 节点，并指向该新置的工作目录 i 节点。返回 0。
        iput(current->pwd);
        current->pwd = inode;
        return (0);
}

//// 改变根目录系统调用函数。
// 将指定的路径名改为根目录'/'。
// 如果操作成功则返回 0，否则返回出错码。
int sys_chroot(const char * filename)
{
        struct m_inode * inode;

// 如果文件名对应的 i 节点不存在，则返回出错码。
        if (!(inode=namei(filename)))
                return -ENOENT;
// 如果该 i 节点不是目录的 i 节点，则释放该节点，返回出错码。
        if (!S_ISDIR(inode->i_mode)) {
                iput(inode);
                return -ENOTDIR;
        }
// 释放当前进程的根目录 i 节点，并重置为这里指定目录名的 i 节点，返回 0。
        iput(current->root);
        current->root = inode;
        return (0);
}

//// 修改文件属性系统调用函数。
// 参数 filename 是文件名，mode 是新的文件属性。
// 若操作成功则返回 0，否则返回出错码。
int sys_chmod(const char * filename,int mode)
{
        struct m_inode * inode;

// 如果文件名对应的 i 节点不存在，则返回出错码。
        if (!(inode=namei(filename)))
                return -ENOENT;
// 如果当前进程的有效用户 id 不等于文件 i 节点的用户 id，并且当前进程不是超级用户，则释放该
// 文件 i 节点，返回出错码。
        if ((current->euid != inode->i_uid) && !suser()) {
                iput(inode);
                return -EACCES;
        }
// 重新设置 i 节点的文件属性，并置该 i 节点已修改标志。释放该 i 节点，返回 0。
        inode->i_mode = (mode & 07777) | (inode->i_mode & ~07777);
```

```
        inode->i_dirt = 1;
        iput(inode);
        return 0;
}

//// 修改文件宿主系统调用函数。
// 参数 filename 是文件名，uid 是用户标识符(用户 id)，gid 是组 id。
// 若操作成功则返回 0，否则返回出错码。
int sys_chown(const char * filename,int uid,int gid)
{
        struct m_inode * inode;

// 如果文件名对应的 i 节点不存在，则返回出错码。
        if (!(inode=namei(filename)))
                return -ENOENT;
// 若当前进程不是超级用户，则释放该 i 节点，返回出错码。
        if (!suser()) {
                iput(inode);
                return -EACCES;
        }
// 设置文件对应 i 节点的用户 id 和组 id，并置 i 节点已经修改标志，释放该 i 节点，返回 0。
        inode->i_uid=uid;
        inode->i_gid=gid;
        inode->i_dirt=1;
        iput(inode);
        return 0;
}

//// 打开（或创建）文件系统调用函数。
// 参数 filename 是文件名，flag 是打开文件标志：只读 O_RDONLY、只写 O_WRONLY 或读写 O_RDWR，
// 以及 O_CREAT、O_EXCL、O_APPEND 等其它一些标志的组合，若本函数创建了一个新文件，则 mode
// 用于指定使用文件的许可属性，这些属性有 S_IRWXU(文件宿主具有读、写和执行权限)、S_IRUSR
// (用户具有读文件权限)、S_IRWXG(组成员具有读、写和执行权限)等等。对于新创建的文件，这些
// 属性只应用于将来对文件的访问，创建了只读文件的打开调用也将返回一个可读写的文件句柄。
// 若操作成功则返回文件句柄(文件描述符)，否则返回出错码。(参见 sys/stat.h, fcntl.h)
int sys_open(const char * filename,int flag,int mode)
{
        struct m_inode * inode;
        struct file * f;
        int i,fd;

// 将用户设置的模式与进程的模式屏蔽码相与，产生许可的文件模式。
        mode &= 0777 & ~current->umask;
// 搜索进程结构中文件结构指针数组，查找一个空闲项，若已经没有空闲项，则返回出错码。
        for(fd=0 ; fd<NR_OPEN ; fd++)
                if (!current->filp[fd])
                        break;
        if (fd>=NR_OPEN)
                return -EINVAL;
// 设置执行时关闭文件句柄位图，复位对应比特位。
        current->close_on_exec &= ~(1<<fd);
// 令 f 指向文件表数组开始处。搜索空闲文件结构项(句柄引用计数为 0 的项)，若已经没有空闲
// 文件表结构项，则返回出错码。
```

```
        f=0+file_table;
        for (i=0 ; i<NR_FILE ; i++,f++)
                if (!f->f_count) break;
        if (i>=NR_FILE)
                return -EINVAL;
// 让进程的对应文件句柄的文件结构指针指向搜索到的文件结构，并令句柄引用计数递增 1。
        (current->filp[fd]=f)->f_count++;
// 调用函数执行打开操作，若返回值小于 0，则说明出错，释放刚申请到的文件结构，返回出错码。
        if ((i=open_namei(filename,flag,mode,&inode))<0) {
                current->filp[fd]=NULL;
                f->f_count=0;
                return i;
        }
/* ttys are somewhat special (ttyxx major==4, tty major==5) */
/* ttys 有些特殊（ttyxx 主号==4，tty 主号==5）*/
// 如果是字符设备文件，那么如果设备号是 4 的话，则设置当前进程的 tty 号为该 i 节点的子设备号。
// 并设置当前进程 tty 对应的 tty 表项的父进程组号等于进程的父进程组号。
        if (S_ISCHR(inode->i_mode))
                if (MAJOR(inode->i_zone[0])==4) {
                        if (current->leader && current->tty<0) {
                                current->tty = MINOR(inode->i_zone[0]);
                                tty_table[current->tty].pgrp = current->pgrp;
                        }
// 否则如果该字符文件设备号是 5 的话，若当前进程没有 tty，则说明出错，释放 i 节点和申请到的
// 文件结构，返回出错码。
                } else if (MAJOR(inode->i_zone[0])==5)
                        if (current->tty<0) {
                                iput(inode);
                                current->filp[fd]=NULL;
                                f->f_count=0;
                                return -EPERM;
                        }
/* Likewise with block-devices: check for floppy_change */
/* 同样对于块设备文件：需要检查盘片是否被更换 */
// 如果打开的是块设备文件，则检查盘片是否更换，若更换则需要是高速缓冲中对应该设备的所有
// 缓冲块失效。
        if (S_ISBLK(inode->i_mode))
                check_disk_change(inode->i_zone[0]);
// 初始化文件结构。置文件结构属性和标志，置句柄引用计数为 1，设置 i 节点字段，文件读写指针
// 初始化为 0。返回文件句柄。
        f->f_mode = inode->i_mode;
        f->f_flags = flag;
        f->f_count = 1;
        f->f_inode = inode;
        f->f_pos = 0;
        return (fd);
}

//// 创建文件系统调用函数。
// 参数 pathname 是路径名，mode 与上面的 sys_open()函数相同。
// 成功则返回文件句柄，否则返回出错码。
int sys_creat(const char * pathname, int mode)
{
```

```
        return sys_open(pathname, O_CREAT | O_TRUNC, mode);
}

// 关闭文件系统调用函数。
// 参数 fd 是文件句柄。
// 成功则返回 0，否则返回出错码。
int sys_close(unsigned int fd)
{
        struct file * filp;

// 若文件句柄值大于程序同时能打开的文件数，则返回出错码。
        if (fd >= NR_OPEN)
                return -EINVAL;
// 复位进程的执行时关闭文件句柄位图对应位。
        current->close_on_exec &= ~(1<<fd);
// 若该文件句柄对应的文件结构指针是 NULL，则返回出错码。
        if (!(filp = current->filp[fd]))
                return -EINVAL;
// 置该文件句柄的文件结构指针为 NULL。
        current->filp[fd] = NULL;
// 若在关闭文件之前，对应文件结构中的句柄引用计数已经为 0，则说明内核出错，死机。
        if (filp->f_count == 0)
                panic("Close: file count is 0");
// 否则将对应文件结构的句柄引用计数减 1，如果还不为 0，则返回 0（成功）。若已等于 0，说明该
// 文件已经没有句柄引用，则释放该文件 i 节点，返回 0。
        if (--filp->f_count)
                return (0);
        iput(filp->f_inode);
        return (0);
}

```

9.15.3 其它信息

9.16 exec.c 程序

9.16.1 功能描述

本源程序实现对二进制可执行文件和 shell 脚本文件的加载与执行。其中主要的函数是函数 do_execve()，它是系统中断调用(int 0x80)功能号__NR_execve()调用的 C 处理函数，是 exec()函数簇的主要实现函数。其主要功能为：

- 执行对参数和环境参数空间页面的初始化操作 -- 设置初始空间起始指针；初始化空间页面指针数组为(NULl)；根据执行文件名取执行对象的 I 节点；计算参数个数和环境变量个数；检查文件类型，执行权限；
- 根据执行文件开始部分的头数据结构，对其中信息进行处理 -- 根据被执行文件 I 节点读取文件头部信息；若是 Shell 脚本程序（第一行以#!开始），则分析 Shell 程序名及其参数，并以被执行文件作为参数执行该执行的 Shell 程序；执行根据文件的幻数以及段长度等信息判断是否可执行；

- 对当前调用进程进行运行新文件前初始化操作 -- 指向新执行文件的 I 节点；复位信号处理句柄；根据头结构信息设置局部描述符基址和段长；设置参数和环境参数页面指针；修改进行各执行字段内容；
- 替换堆栈上原调用 execve()程序的返回地址为新执行程序运行地址，运行新加载的程序。

execve()函数有大量对参数和环境空间的处理操作,参数和环境空间共可有 MAX_ARG_PAGES 个页面，总长度可达 128kB 字节。在该空间中存放数据的方式类似于堆栈操作，即是从假设的 128kB 空间末端处逆向开始存放参数或环境变量字符串的。在初始时，程序定义了一个指向该空间末端(128kB-4 字节)处空间内偏移值 p，该偏移值随着存放数据的增多而后退，由图中可以看出，p 明确地指出了当前参数环境空间中还剩余多少可用空间。在分析程序中 copy_string()函数时，可参照此图。

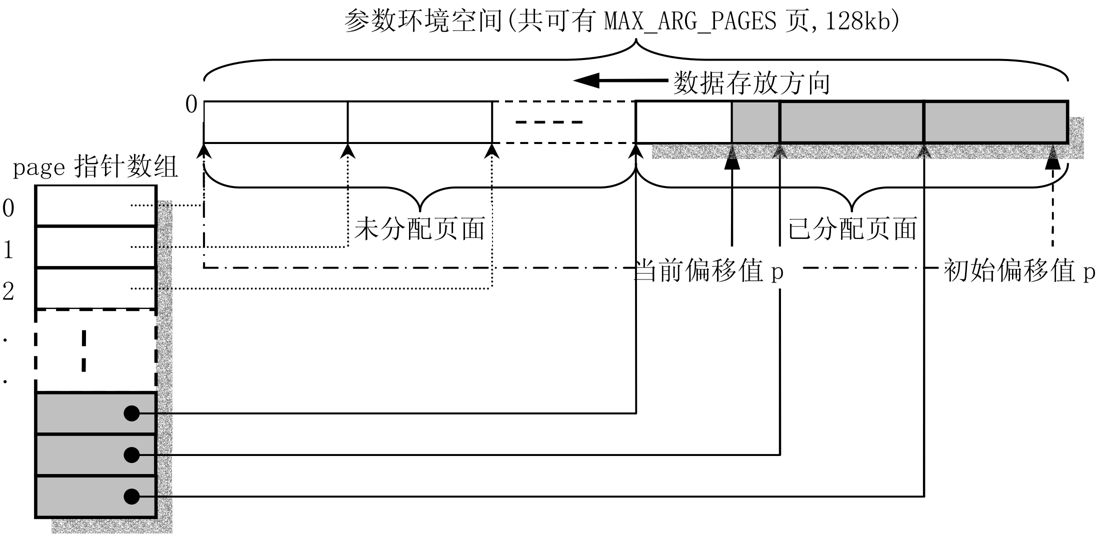


图 参数和环境变量字符串空间

create_tables()函数用于根据给定的当前堆栈指针值 p 以及参数变量个数值 argc 和环境变量个数 envc，在新的程序堆栈中创建环境和参数变量指针表，并返回此时的堆栈指针值 sp。创建完毕后堆栈指针表的形式见下图所示。

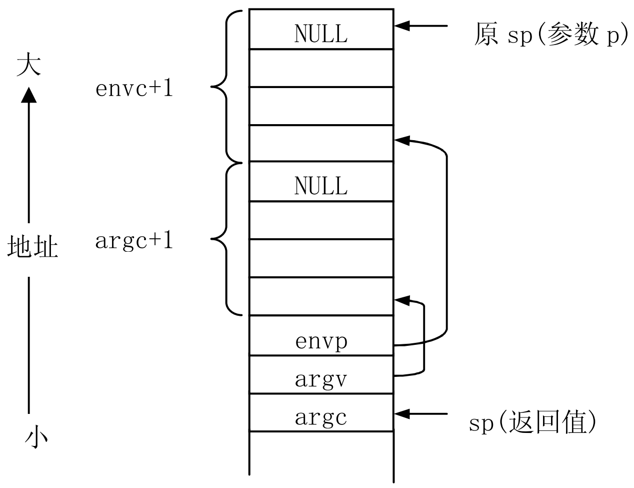


图 新程序堆栈中指针表示意图

9.16.2 代码注释

列表 linux/fs/exec.c 程序

```
/*
 *  linux/fs/exec.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * #!-checking implemented by tytso.
 */
/*
 * #!开始的程序检测部分是由 tytso 实现的。
 */

/*
 * Demand-loading implemented 01.12.91 - no need to read anything but
 * the header into memory. The inode of the executable is put into
 * "current->executable", and page faults do the actual loading. Clean.
 *
 * Once more I can proudly say that linux stood up to being changed: it
 * was less than 2 hours work to get demand-loading completely implemented.
 */
/*
 * 需求时加载是于 1991.12.1 实现的 - 只需将执行文件头部分读进内存而无须
 * 将整个执行文件都加载进内存。执行文件的 i 节点被放在当前进程的可执行字段中
 * ("current->executable")，而页异常会进行执行文件的实际加载操作以及清理工作。
 *
 * 我可以再一次自豪地说，linux 经得起修改：只用了不到 2 小时的工作时间就完全
 * 实现了需求加载处理。
 */

#include <errno.h>         // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
#include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
#include <sys/stat.h>      // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。
#include <a.out.h>         // a.out 头文件。定义了 a.out 执行文件格式和一些宏。

#include <linux/fs.h>      // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h>  // 内核头文件。含有一些内核常用函数的原形定义。
#include <linux/mm.h>      // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
#include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

extern int sys_exit(int exit_code);   // 程序退出系统调用。
extern int sys_close(int fd);         // 文件关闭系统调用。

/*
 * MAX_ARG_PAGES defines the number of pages allocated for arguments
 * and envelope for the new program. 32 should suffice, this gives
 * a maximum env+arg of 128kB !
 */
```

```
/*
 * MAX_ARG_PAGES 定义了新程序分配给参数和环境变量使用的内存最大页数。
 * 32 页内存应该足够了，这使得环境和参数(env+arg)空间的总合达到 128kB!
 */
#define MAX_ARG_PAGES 32

/*
 * create_tables() parses the env- and arg-strings in new user
 * memory and creates the pointer tables from them, and puts their
 * addresses on the "stack", returning the new stack pointer value.
 */
/*
 * create_tables()函数在新用户内存中解析环境变量和参数字符串，由此
 * 创建指针表，并将它们的地址放到"堆栈"上，然后返回新栈的指针值。
 */
//// 在新用户堆栈中创建环境和参数变量指针表。
// 参数：p - 以数据段为起点的参数和环境信息偏移指针；argc - 参数个数；envc -环境变量数。
// 返回：堆栈指针。
static unsigned long * create_tables(char * p,int argc,int envc)
{
        unsigned long *argv,*envp;
        unsigned long * sp;

// 堆栈指针是以 4 字节（1 节）为边界寻址的，因此这里让 sp 为 4 的整数倍。
        sp = (unsigned long *) (0xfffffffc & (unsigned long) p);
// sp 向下移动，空出环境参数占用的空间个数，并让环境参数指针 envp 指向该处。
        sp -= envc+1;
        envp = sp;
// sp 向下移动，空出命令行参数指针占用的空间个数，并让 argv 指针指向该处。
// 下面指针加 1，sp 将递增指针宽度字节值。
        sp -= argc+1;
        argv = sp;
// 将环境参数指针 envp 和命令行参数指针以及命令行参数个数压入堆栈。
        put_fs_long((unsigned long)envp,--sp);
        put_fs_long((unsigned long)argv,--sp);
        put_fs_long((unsigned long)argc,--sp);
// 将命令行各参数指针放入前面空出来的相应地方，最后放置一个 NULL 指针。
        while (argc-->0) {
                put_fs_long((unsigned long) p,argv++);
                while (get_fs_byte(p++)) /* nothing */ ;  // p 指针前移 4 字节。
        }
        put_fs_long(0,argv);
// 将环境变量各指针放入前面空出来的相应地方，最后放置一个 NULL 指针。
        while (envc-->0) {
                put_fs_long((unsigned long) p,envp++);
                while (get_fs_byte(p++)) /* nothing */ ;
        }
        put_fs_long(0,envp);
        return sp;                       // 返回构造的当前新堆栈指针。
}

/*
 * count() counts the number of arguments/envelopes
```

```
 */
/*
 * count()函数计算命令行参数/环境变量的个数。
 */
//// 计算参数个数。
// 参数：argv - 参数指针数组，最后一个指针项是 NULL。
// 返回：参数个数。
static int count(char ** argv)
{
        int i=0;
        char ** tmp;

        if (tmp = argv)
                while (get_fs_long((unsigned long *) (tmp++)))
                        i++;

        return i;
}

/*
 * 'copy_string()' copies argument/envelope strings from user
 * memory to free pages in kernel mem. These are in a format ready
 * to be put directly into the top of new user memory.
 *
 * Modified by TYT, 11/24/91 to add the from_kmem argument, which specifies
 * whether the string and the string array are from user or kernel segments:
 *
 * from_kmem     argv *        argv **
 *    0          user space    user space
 *    1          kernel space  user space
 *    2          kernel space  kernel space
 *
 * We do this by playing games with the fs segment register.  Since it
 * it is expensive to load a segment register, we try to avoid calling
 * set_fs() unless we absolutely have to.
 */
/*
 * 'copy_string()'函数从用户内存空间拷贝参数和环境字符串到内核空闲页面内存中。
 * 这些已具有直接放到新用户内存中的格式。
 *
 * 由 TYT(Tytso)于 1991.12.24 日修改，增加了 from_kmem 参数，该参数指明了字符串或
 * 字符串数组是来自用户段还是内核段。
 *
 * from_kmem     argv *        argv **
 *    0          用户空间      用户空间
 *    1          内核空间      用户空间
 *    2          内核空间      内核空间
 *
 * 我们是通过巧妙处理 fs 段寄存器来操作的。由于加载一个段寄存器代价太大，所以
 * 我们尽量避免调用 set_fs()，除非实在必要。
 */
//// 复制指定个数的参数字符串到参数和环境空间。
// 参数：argc - 欲添加的参数个数；argv - 参数指针数组；page - 参数和环境空间页面指针数组。
```

```
//        p -在参数表空间中的偏移指针，始终指向已复制串的头部；from_kmem - 字符串来源标志。
// 在 do_execve()函数中，p 初始化为指向参数表(128kB)空间的最后一个长字处，参数字符串
// 是以堆栈操作方式逆向往其中复制存放的，因此 p 指针会始终指向参数字符串的头部。
// 返回：参数和环境空间当前头部指针。
static unsigned long copy_strings(int argc,char ** argv,unsigned long *page,
                unsigned long p, int from_kmem)
{
        char *tmp, *pag;
        int len, offset = 0;
        unsigned long old_fs, new_fs;

        if (!p)
                return 0;       /* bullet-proofing */  /* 偏移指针验证 */
// 取 ds 寄存器值到 new_fs，并保存原 fs 寄存器值到 old_fs。
        new_fs = get_ds();
        old_fs = get_fs();
// 如果字符串和字符串数组来自内核空间，则设置 fs 段寄存器指向内核数据段（ds）。
        if (from_kmem==2)
                set_fs(new_fs);
// 循环处理各个参数，从最后一个参数逆向开始复制，复制到指定偏移地址处。
        while (argc-- > 0) {
// 如果字符串在用户空间而字符串数组在内核空间，则设置 fs 段寄存器指向内核数据段（ds）。
                if (from_kmem == 1)
                        set_fs(new_fs);
// 从最后一个参数开始逆向操作，取 fs 段中最后一参数指针到 tmp，如果为空，则出错死机。
                if (!(tmp = (char *)get_fs_long(((unsigned long *)argv)+argc)))
                        panic("argc is wrong");
// 如果字符串在用户空间而字符串数组在内核空间，则恢复 fs 段寄存器原值。
                if (from_kmem == 1)
                        set_fs(old_fs);
// 计算该参数字符串长度 len，并使 tmp 指向该参数字符串末端。
                len=0;          /* remember zero-padding */
                do {            /* 我们知道串是以 NULL 字节结尾的 */
                        len++;
                } while (get_fs_byte(tmp++));
// 如果该字符串长度超过此时参数和环境空间中还剩余的空闲长度，则恢复 fs 段寄存器并返回 0。
                if (p-len < 0) {        /* this shouldn't happen - 128kB */
                        set_fs(old_fs); /* 不会发生-因为有 128kB 的空间 */
                        return 0;
                }
// 复制 fs 段中当前指定的参数字符串，是从该字符串尾逆向开始复制。
                while (len) {
                        --p; --tmp; --len;
// 函数刚开始执行时，偏移变量 offset 被初始化为 0，因此若 offset-1<0，说明是首次复制字符串，
// 则令其等于 p 指针在页面内的偏移值，并申请空闲页面。
                        if (--offset < 0) {
                                offset = p % PAGE_SIZE;
// 如果字符串和字符串数组在内核空间，则恢复 fs 段寄存器原值。
                                if (from_kmem==2)
                                        set_fs(old_fs);
// 如果当前偏移值 p 所在的串空间页面指针数组项 page[p/PAGE_SIZE]==0，表示相应页面还不存在，
// 则需申请新的内存空闲页面，将该页面指针填入指针数组，并且也使 pag 指向该新页面，若申请不
// 到空闲页面则返回 0。
```

```
                                        if (!(pag = (char *) page[p/PAGE_SIZE]) &&
                                            !(pag = (char *) page[p/PAGE_SIZE] =
                                              (unsigned long *) get_free_page()))
                                                return 0;
// 如果字符串和字符串数组来自内核空间，则设置 fs 段寄存器指向内核数据段（ds）。
                                        if (from_kmem==2)
                                                set_fs(new_fs);

                                }
// 从 fs 段中复制参数字符串中一字节到 pag+offset 处。
                                *(pag + offset) = get_fs_byte(tmp);
                        }
                }
// 如果字符串和字符串数组在内核空间，则恢复 fs 段寄存器原值。
        if (from_kmem==2)
                set_fs(old_fs);
// 最后，返回参数和环境空间中已复制参数信息的头部偏移值。
        return p;
}

//// 修改局部描述符表中的描述符基址和段限长，并将参数和环境空间页面放置在数据段末端。
// 参数：text_size - 执行文件头部中 a_text 字段给出的代码段长度值；
//       page - 参数和环境空间页面指针数组。
// 返回：数据段限长值(64MB)。
static unsigned long change_ldt(unsigned long text_size,unsigned long * page)
{
        unsigned long code_limit,data_limit,code_base,data_base;
        int i;

// 根据执行文件头部 a_text 值，计算以页面长度为边界的代码段限长。并设置数据段长度为 64MB。
        code_limit = text_size+PAGE_SIZE -1;
        code_limit &= 0xFFFFF000;
        data_limit = 0x4000000;
// 取当前进程中局部描述符表代码段描述符中代码段基址，代码段基址与数据段基址相同。
        code_base = get_base(current->ldt[1]);
        data_base = code_base;
// 重新设置局部表中代码段和数据段描述符的基址和段限长。
        set_base(current->ldt[1],code_base);
        set_limit(current->ldt[1],code_limit);
        set_base(current->ldt[2],data_base);
        set_limit(current->ldt[2],data_limit);
/* make sure fs points to the NEW data segment */
/* 要确信 fs 段寄存器已指向新的数据段 */
// fs 段寄存器中放入局部表数据段描述符的选择符(0x17)。
        __asm__("pushl $0x17\n\tpop %%fs"::);
// 将参数和环境空间已存放数据的页面（共可有 MAX_ARG_PAGES 页，128kB）放到数据段线性地址的
// 末端。是调用函数 put_page()进行操作的（mm/memory.c, 197）。
        data_base += data_limit;
        for (i=MAX_ARG_PAGES-1 ; i>=0 ; i--) {
                data_base -= PAGE_SIZE;
                if (page[i])                            // 如果该页面存在，
                        put_page(page[i],data_base);    // 就放置该页面。
        }
```

```
176         return data_limit;                          // 最后返回数据段限长(64MB)。
177 }
178
179 /*
180  * 'do_execve()' executes a new program.
181  */
    /*
     * 'do_execve()'函数执行一个新程序。
     */
    //// execve()系统中断调用函数。加载并执行子进程（其它程序）。
    // 该函数系统中断调用(int 0x80)功能号__NR_execve 调用的函数。
    // 参数：eip - 指向堆栈中调用系统中断的程序代码指针 eip 处，参见 kernel/system_call.s 程序
    // 开始部分的说明；tmp - 系统中断调用本函数时的返回地址，无用；
    //      filename - 被执行程序文件名；argv - 命令行参数指针数组；envp - 环境变量指针数组。
    // 返回：如果调用成功，则不返回；否则设置出错号，并返回-1。
182 int do_execve(unsigned long * eip,long tmp,char * filename,
183         char ** argv, char ** envp)
184 {
185         struct m_inode * inode;                 // 内存中 I 节点指针结构变量。
186         struct buffer_head * bh;                // 高速缓存块头指针。
187         struct exec ex;                         // 执行文件头部数据结构变量。
188         unsigned long page[MAX_ARG_PAGES];      // 参数和环境字符串空间的页面指针数组。
189         int i,argc,envc;
190         int e_uid, e_gid;                       // 有效用户 id 和有效组 id。
191         int retval;                             // 返回值。
192         int sh_bang = 0;                        // 控制是否需要执行脚本处理代码。
    // 参数和环境字符串空间中的偏移指针，初始化为指向该空间的最后一个长字处。
193         unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4;
194
    // eip[1]中是原代码段寄存器 cs，其中的选择符不可以是内核段选择符，也即内核不能调用本函数。
195         if ((0xffff & eip[1]) != 0x000f)
196                 panic("execve called from supervisor mode");
    // 初始化参数和环境串空间的页面指针数组（表）。
197         for (i=0 ; i<MAX_ARG_PAGES ; i++)       /* clear page-table */
198                 page[i]=0;
    // 取可执行文件的对应 i 节点号。
199         if (!(inode=namei(filename)))           /* get executables inode */
200                 return -ENOENT;
    // 计算参数个数和环境变量个数。
201         argc = count(argv);
202         envc = count(envp);
203
    // 执行文件必须是常规文件。若不是常规文件则置出错返回码，跳转到 exec_error2(第 347 行)。
204 restart_interp:
205         if (!S_ISREG(inode->i_mode)) {  /* must be regular file */
206                 retval = -EACCES;
207                 goto exec_error2;
208         }
    // 检查被执行文件的执行权限。根据其属性(对应 i 节点的 uid 和 gid)，看本进程是否有权执行它。
209         i = inode->i_mode;
210         e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
211         e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;
212         if (current->euid == inode->i_uid)
```

```
                i >>= 6;
        else if (current->egid == inode->i_gid)
                i >>= 3;
        if (!(i & 1) &&
            !((inode->i_mode & 0111) && suser())) {
                retval = -ENOEXEC;
                goto exec_error2;
        }
// 读取执行文件的第一块数据到高速缓冲区，若出错则置出错码，跳转到 exec_error2 处去处理。
        if (!(bh = bread(inode->i_dev,inode->i_zone[0]))) {
                retval = -EACCES;
                goto exec_error2;
        }
// 下面对执行文件的头结构数据进行处理，首先让 ex 指向执行头部分的数据结构。
        ex = *((struct exec *) bh->b_data);     /* read exec-header */ /* 读取执行头部分 */
// 如果执行文件开始的两个字节为'#!'，并且 sh_bang 标志没有置位，则处理脚本文件的执行。
        if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) {
                /*
                 * This section does the #! interpretation.
                 * Sorta complicated, but hopefully it will work.  -TYT
                 */
                /*
                 * 这部分处理对'#!'的解释，有些复杂，但希望能工作。-TYT
                 */

                char buf[1023], *cp, *interp, *i_name, *i_arg;
                unsigned long old_fs;

// 复制执行程序头一行字符'#!'后面的字符串到 buf 中，其中含有脚本处理程序名。
                strncpy(buf, bh->b_data+2, 1022);
// 释放高速缓冲块和该执行文件 i 节点。
                brelse(bh);
                iput(inode);
// 取第一行内容，并删除开始的空格、制表符。
                buf[1022] = '\0';
                if (cp = strchr(buf, '\n')) {
                        *cp = '\0';
                        for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
                }
// 若该行没有其它内容，则出错。置出错码，跳转到 exec_error1 处。
                if (!cp || *cp == '\0') {
                        retval = -ENOEXEC; /* No interpreter name found */
                        goto exec_error1;
                }
// 否则就得到了开头是脚本解释执行程序名称的一行内容。
                interp = i_name = cp;
// 下面分析该行。首先取第一个字符串，其应该是脚本解释程序名，iname 指向该名称。
                i_arg = 0;
                for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++) {
                        if (*cp == '/')
                                i_name = cp+1;
                }
// 若文件名后还有字符，则应该是参数串，令 i_arg 指向该串。
```

```
                if (*cp) {
                        *cp++ = '\0';
                        i_arg = cp;
                }
                /*
                 * OK, we've parsed out the interpreter name and
                 * (optional) argument.
                 */
                /*
                 * OK，我们已经解析出解释程序的文件名以及(可选的)参数。
                 */
// 若 sh_bang 标志没有设置，则设置它，并复制指定个数的环境变量串和参数串到参数和环境空间中。
                if (sh_bang++ == 0) {
                        p = copy_strings(envc, envp, page, p, 0);
                        p = copy_strings(--argc, argv+1, page, p, 0);
                }
                /*
                 * Splice in (1) the interpreter's name for argv[0]
                 *           (2) (optional) argument to interpreter
                 *           (3) filename of shell script
                 *
                 * This is done in reverse order, because of how the
                 * user environment and arguments are stored.
                 */
                /*
                 * 拼接 (1) argv[0]中放解释程序的名称
                 *      (2) (可选的)解释程序的参数
                 *      (3) 脚本程序的名称
                 *
                 * 这是以逆序进行处理的，是由于用户环境和参数的存放方式造成的。
                 */
// 复制脚本程序文件名到参数和环境空间中。
                p = copy_strings(1, &filename, page, p, 1);
// 复制解释程序的参数到参数和环境空间中。
                argc++;
                if (i_arg) {
                        p = copy_strings(1, &i_arg, page, p, 2);
                        argc++;
                }
// 复制解释程序文件名到参数和环境空间中。若出错，则置出错码，跳转到 exec_error1。
                p = copy_strings(1, &i_name, page, p, 2);
                argc++;
                if (!p) {
                        retval = -ENOMEM;
                        goto exec_error1;
                }
                /*
                 * OK, now restart the process with the interpreter's inode.
                 */
                /*
                 * OK，现在使用解释程序的 i 节点重启进程。
                 */
// 保留原 fs 段寄存器（原指向用户数据段），现置其指向内核数据段。
```

```
288                 old_fs = get_fs();
289                 set_fs(get_ds());
    // 取解释程序的 i 节点，并跳转到 restart_interp 处重新处理。
290                 if (!(inode=namei(interp))) { /* get executables inode */
291                         set_fs(old_fs);
292                         retval = -ENOENT;
293                         goto exec_error1;
294                 }
295                 set_fs(old_fs);
296                 goto restart_interp;
297         }
    // 释放该缓冲区。
298         brelse(bh);
    // 下面对执行头信息进行处理。
    // 对于下列情况，将不执行程序：如果执行文件不是需求页可执行文件(ZMAGIC)、或者代码重定位部分
    // 长度 a_trsize 不等于 0、或者数据重定位信息长度不等于 0、或者代码段+数据段+堆段长度超过 50MB、
    // 或者 i 节点表明的该执行文件长度小于代码段+数据段+符号表长度+执行头部分长度的总和。
299         if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
300                 ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
301                 inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
302                 retval = -ENOEXEC;
303                 goto exec_error2;
304         }
    // 如果执行文件执行头部分长度不等于一个内存块大小（1024 字节），也不能执行。转 exec_error2。
305         if (N_TXTOFF(ex) != BLOCK_SIZE) {
306                 printk("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename);
307                 retval = -ENOEXEC;
308                 goto exec_error2;
309         }
    // 如果 sh_bang 标志没有设置，则复制指定个数的环境变量字符串和参数到参数和环境空间中。
    // 若 sh_bang 标志已经设置，则表明是将运行脚本程序，此时环境变量页面已经复制，无须再复制。
310         if (!sh_bang) {
311                 p = copy_strings(envc,envp,page,p,0);
312                 p = copy_strings(argc,argv,page,p,0);
    // 如果 p=0，则表示环境变量与参数空间页面已经被占满，容纳不下了。转至出错处理处。
313                 if (!p) {
314                         retval = -ENOMEM;
315                         goto exec_error2;
316                 }
317         }
318 /* OK, This is the point of no return */
    /* OK，下面开始就没有返回的地方了 */
    // 如果原程序也是一个执行程序，则释放其 i 节点，并让进程 executable 字段指向新程序 i 节点。
319         if (current->executable)
320                 iput(current->executable);
321         current->executable = inode;
    // 清复位所有信号处理句柄。但对于 SIG_IGN 句柄不能复位，因此在 322 与 323 行之间需添加一条
    // if 语句：if (current->sa[I].sa_handler != SIG_IGN)。这是源代码中的一个 bug。
322         for (i=0 ; i<32 ; i++)
323                 current->sigaction[i].sa_handler = NULL;
    // 根据执行时关闭(close_on_exec)文件句柄位图标志，关闭指定的打开文件，并复位该标志。
324         for (i=0 ; i<NR_OPEN ; i++)
325                 if ((current->close_on_exec>>i)&1)
```

```
                        sys_close(i);
        current->close_on_exec = 0;
// 根据指定的基地址和限长，释放原来程序代码段和数据段所对应的内存页表指定的内存块及页表本身。
        free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
        free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
// 如果“上次任务使用了协处理器”指向的是当前进程，则将其置空，并复位使用了协处理器的标志。
        if (last_task_used_math == current)
                last_task_used_math = NULL;
        current->used_math = 0;
// 根据 a_text 修改局部表中描述符基址和段限长，并将参数和环境空间页面放置在数据段末端。
// 执行下面语句之后，p 此时是以数据段起始处为原点的偏移值，仍指向参数和环境空间数据开始处，
// 也即转换成为堆栈的指针。
        p += change_ldt(ex.a_text,page)-MAX_ARG_PAGES*PAGE_SIZE;
// create_tables()在新用户堆栈中创建环境和参数变量指针表，并返回该堆栈指针。
        p = (unsigned long) create_tables((char *)p,argc,envc);
// 修改当前进程各字段为新执行程序的信息。令进程代码段尾值字段 end_code = a_text；令进程数据
// 段尾字段 end_data = a_data + a_text；令进程堆结尾字段 brk = a_text + a_data + a_bss。
        current->brk = ex.a_bss +
                (current->end_data = ex.a_data +
                (current->end_code = ex.a_text));
// 设置进程堆栈开始字段为堆栈指针所在的页面，并重新设置进程的用户 id 和组 id。
        current->start_stack = p & 0xfffff000;
        current->euid = e_uid;
        current->egid = e_gid;
// 初始化一页 bss 段数据，全为零。
        i = ex.a_text+ex.a_data;
        while (i&0xfff)
                put_fs_byte(0,(char *) (i++));
// 将原调用系统中断的程序在堆栈上的代码指针替换为指向新执行程序的入口点，并将堆栈指针替换
// 为新执行程序的堆栈指针。返回指令将弹出这些堆栈数据并使得 CPU 去执行新的执行程序，因此不会
// 返回到原调用系统中断的程序中去了。
        eip[0] = ex.a_entry;            /* eip, magic happens :-) */ /* eip，魔法起作用了*/
        eip[3] = p;                     /* stack pointer */           /* esp，堆栈指针 */
        return 0;
exec_error2:
        iput(inode);
exec_error1:
        for (i=0 ; i<MAX_ARG_PAGES ; i++)
                free_page(page[i]);
        return(retval);
}
```

9.16.3 其它信息

9.16.3.1 a.out 执行文件格式

Linux 内核 0.11 版仅支持 a.out(Assembley & link editor output)执行文件格式，虽然这种格式目前已经渐渐不用，而使用功能更为齐全的 ELF（Executable and Link Format）格式，但是由于其简单性，作为学习入门的材料正好比较适用。下面全面介绍一下 a.out 格式。

在头文件<a.out.h>中申明了三个数据结构以及一些宏函数。这些数据结构描述了系统上可执行的机器码文件（二进制文件）。

一个执行文件共可有七个部分（七节）组成。按照顺序，这些部分是：

执行头部分(exec header)
执行文件头部分。该部分中含有一些参数，内核使用这些参数将执行文件加载到内存中并执行，而链接程序(ld)使用这些参数将一些二进制目标文件组合成一个可执行文件。这是唯一必要的组成部分。

代码段部分(text segment)
含有程序执行使被加载到内存中的指令代码和相关数据。可以以只读形式进行加载。

数据段部分(data segment)
这部分含有已经初始化过的数据，总是被加载到可读写的内存中。

代码重定位部分(text relocations)
这部分含有供链接程序使用的记录数据。在组合二进制目标文件时用于定位代码段中的指针或地址。

数据重定位部分(data relocations)
与代码重定位部分的作用类似，但是是用于数据段中指针的重定位。

符号表部分(simbol table)
这部分同样含有供链接程序使用的记录数据，用于在二进制目标文件之间对命名的变量和函数（符号）进行交叉引用。

字符串表部分(string table)
该部分含有与符号名相对应的字符串。

每个二进制执行文件均以一个执行数据结构（exec structure）开始。该数据结构的形式如下：

```
struct exec {
        unsigned long a_midmag;
        unsigned long a_text;
        unsigned long a_data;
        unsigned long a_bss;
        unsigned long a_syms;
        unsigned long a_entry;
        unsigned long a_trsize;
        unsigned long a_drsize;
};
```

各个字段的功能如下：

a_midmag　该字段含有被 N_GETFLAG()、N_GETMID 和 N_GETMAGIC()访问的子部分，是由链接程序

在 运行时加载到进程地址空间。宏 N_GETMID()用于返回机器标识符(machine-id)，指示
出 二进制文件将在什么机器上运行。N_GETMAGIC()宏指明魔数，它唯一地确定了二进制
执 行文件与其它加载的文件之间的区别。字段中必须包含以下值之一：

OMAGIC 表示代码和数据段紧随在执行头后面并且是连续存放的。内核将代码和数据
段 都加载到可读写内存中。

NMAGIC 同 OMAGIC 一样，代码和数据段紧随在执行头后面并且是连续存放的。然而内核将代码加载到了只读内存中，并把数据段加载到代码段后下一页可读
写内存 边界开始。

ZMAGIC 内核在必要时从二进制执行文件中加载独立的页面。执行头部、代码段和数
据 段都被链接程序处理成多个页面大小的块。内核加载的代码页面时只读的，
而 数据段的页面是可写的。

a_text 该字段含有代码段的长度值，字节数。

a_data 该字段含有数据段的长度值，字节数。

a_bss 含有‘bss 段’的长度，内核用其设置在数据段后初始的 break（brk）。内核在加载程序时，这段可写内存显现出处于数据段后面，并且初始时为全零。

a_syms 含有符号表部分的字节长度值。

a_entry 含有内核将执行文件加载到内存中以后，程序执行起始点的内存地址。

a_trsize 该字段含有代码重定位表的大小，是字节数。

a_drsize 该字段含有数据重定位表的大小，是字节数。

在 a.out.h 头文件中定义了几个宏，这些宏使用 exec 结构来测试一致性或者定位执行文件中各个部分（节）的位置偏移值。这些宏有：

N_BADMAG(exec) 如果 a_magic 字段不能被识别，则返回非零值。

N_TXTOFF(exec) 代码段的起始位置字节偏移值。

N_DATOFF(exec) 数据段的起始位置字节偏移值。

N_DRELOFF(exec) 数据重定位表的起始位置字节偏移值。

N_TRELOFF(exec) 代码重定位表的起始位置字节偏移值。

N_SYMOFF(exec) 符号表的起始位置字节偏移值。

N_STROFF(exec) 字符串表的起始位置字节偏移值。

重定位记录具有标准格式，它使用重定位信息(relocation_info)结构来描述：

```
struct relocation_info {
        int             r_address;
        unsigned int    r_symbolnum : 24,
                        r_pcrel : 1,
                        r_length : 2,
                        r_extern : 1,
                        r_baserel : 1,
                        r_jmptable : 1,
                        r_relative : 1,
                        r_copy : 1;
};
```

该结构中各字段的含义如下：

r_address 该字段含有需要链接程序处理（编辑）的指针的字节偏移值。代码重定位的偏移值是从代码段开始处计数的，数据重定位的偏移值是从数据段开始处计算的。链接程序会将已经存储在该偏移处的值与使用重定位记录计算出的新值相加。

r_symbolnum 该字段含有符号表中一个符号结构的序号值（不是字节偏移值）。链接程序在算出符号的绝对地址以后，就将该地址加到正在进行重定位的指针上。（如果 r_extern 比特位是 0，那么情况就不同，见下面。）

r_pcrel 如果设置了该位，链接程序就认为正在更新一个指针，该指针使用 pc 相关寻址方式，是属于机器码指令部分。当运行程序使用这个被重定位的指针时，该指针的地址被隐式地加到该指针上。

r_length 该字段含有指针长度的 2 的次方值：0 表示 1 字节长，1 表示 2 字节长，2 表示 4 字节长。

r_extern 如果被置位，表示该重定位需要一个外部引用；此时链接程序必须使用一个符号地址来更新相应指针。当该位是 0 时，则重定位是“局部”的；链接程序更新指针以反映各个段加载地址中的变化，而不是反映一个符号值的变化（除非同时设置了 r_baserel，见下面）。在这种情况下，r_symbolnum 字段的内容是一个 n_type 值（见下面）；这类字段告诉链接程序被重定位的指针指向那个段。

r_baserel 如果设置了该位，则 r_symbolnum 字段指定的符号将被重定位成全局偏移表(Global Offset Table)中的一个偏移值。在运行时刻，全局偏移表该偏移处被设置为符号的地址。

r_jmptable 如果被置位，则 r_symbolnum 字段指定的符号将被重定位成过程链接表(Procedure Linkage Table)中的一个偏移值。

r_relative 如果被置位，则说明此重定位与该目标文件将成为其组成部分的映象文件在运行时被加载的地址相关。这类重定位仅在共享目标文件中出现。

r_copy 如果被置位，该重定位记录指定了一个符号，该符号的内容将被复制到 r_address 指定的地方。该复制操作是通过共享目标模块中一个合适的数据项中的运行时刻链接程序完成的。

符号将名称映射为地址（或者更通俗地讲是字符串映射到值）。由于链接程序对地址的调整，一个符号的名称必须用来表示其地址，直到已被赋予一个绝对地址值。符号是由符号表中固定长度的记录以及字符串表中的可变长度名称组成。符号表是 nlist 结构的一个数组，如下所示：

```
struct nlist {
        union {
                char    *n_name;
                long    n_strx;
        } n_un;
        unsigned char   n_type;
        char            n_other;
        short           n_desc;
        unsigned long   n_value;
};
```

其中各字段的含义为：

n_un.n_strx 含有本符号的名称在字符串表中的字节偏移值。当程序使用 nlist()函数访问一个符号表时，该字段被替换为 n_un.n_name 字段，这是内存中字符串的指针。

n_type 用于链接程序确定如何更新符号的值。使用位屏蔽(bitmasks)可以将 n_type 字段分割成三个子字段，对于 N_EXT 类型位置位的符号，链接程序将它们看作是“外部的”符号，并且允许其它二进制目标文件对它们的引用。N_TYPE 屏蔽码用于链接程序感兴趣的比特位：

N_UNDF 一个未定义的符号。链接程序必须在其它二进制目标文件中定位一个具有相
同 名称的外部符号，以确定该符号的绝对数据值。特殊情况下，如果 n_type 字段是非零
值，并且没有二进制文件定义了这个符号，则链接程序在 BSS 段中将该符号解析
为一个 地址，保留长度等于 n_value 的字节。如果符号在多于一个二进制目标文件中都
没有定 义并且这些二进制目标文件对其长度值不一致，则链接程序将选择所有二进制目
标文件 中最大的长度。

N_ABS 一个绝对符号。链接程序不会更新一个绝对符号。

N_TEXT 一个代码符号。该符号的值是代码地址，链接程序在合并二进制目标文件时
会 更新其值。

N_DATA 一个数据符号；与 N_TEXT 类似，但是用于数据地址。对应代码和数据符号的
值不是文件的偏移值而是地址；为了找出文件的偏移，就有必要确定相关部分开
始加载 的地址并减去它，然后加上该部分的偏移。

N_BSS　一个 BSS 符号；与代码或数据符号类似，但在二进制目标文件中没有对应的偏移。

N_FN　一个文件名符号。在合并二进制目标文件时，链接程序会将该符号插入在二进制文件中的符号之前。符号的名称就是给予链接程序的文件名，而其值是二进制文件中首个代码段地址。链接和加载时不需要文件名符号，但对于调式程序非常有用。

N_STAB 屏蔽码用于选择符号调式程序(例如 gdb)感兴趣的位；其值在 stab()中说明。

n_other　该字段按照 n_type 确定的段，提供有关符号重定位操作的符号独立性信息。目前，n_other 字段的最低 4 位含有两个值之一：AUX_FUNC 和 AUX_OBJECT（有关定义参见<link.h>）。AUX_FUNC 将符号与可调用的函数相关，AUX_OBJECT 将符号与数据相关，而不管它们是位于代码段还是数据段。该字段主要用于链接程序 ld，用于动态可执行程序的创建。

n_desc　保留给调式程序使用；链接程序不对其进行处理。不同的调试程序将该字段用作不同的用途。

n_value　含有符号的值。对于代码、数据和 BSS 符号，这是一个地址；对于其它符号（例如调式程序符号），值可以是任意的。

字符串表是由长度为 u_int32_t 后跟一 null 结尾的符号字符串组成。长度代表整个表的字节大小，所以在 32 位的机器上其最小值（或者是第 1 个字符串的偏移）总是 4。

9.17 namei.c 文件

9.17.1 功能描述

9.17.2 代码注释

列表 linux/fs/namei.c 程序

```
/*
 *  linux/fs/namei.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * Some corrections by tytso.
 */
/*
 * tytso 作了一些纠正。
 */

```

```
#include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。

#include <string.h>       // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
#include <fcntl.h>        // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
#include <errno.h>        // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
#include <const.h>        // 常数符号头文件。目前仅定义了 i 节点中 i_mode 字段的各标志位。
#include <sys/stat.h>     // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。

#define ACC_MODE(x) ("\004\002\006\377"[(x)&O_ACCMODE])    // 访问模式宏(前面是 8 进制数)。

/*
 * comment out this line if you want names > NAME_LEN chars to be
 * truncated. Else they will be disallowed.
 */
 /*
 * 如果想让文件名长度>NAME_NAME 的字符被截掉，就将下面定义注释掉。
 */
/* #define NO_TRUNCATE */

#define MAY_EXEC 1        // 可执行(可进入)。
#define MAY_WRITE 2       // 可写。
#define MAY_READ 4        // 可读。

/*
 *      permission()
 *
 * is used to check for read/write/execute permissions on a file.
 * I don't know if we should look at just the euid or both euid and
 * uid, but that should be easily changed.
 */
/*
 *      permission()
 * 该函数用于检测一个文件的读/写/执行权限。我不知道是否只需检查 euid 还是
 * 需要检查 euid 和 uid 两者，不过这很容易修改。
 */
//// 检测文件访问许可权限。
// 参数：inode - 文件对应的 i 节点；mask - 访问属性屏蔽码。
// 返回：访问许可返回 1，否则返回 0。
static int permission(struct m_inode * inode,int mask)
{
        int mode = inode->i_mode;

/* special case: not even root can read/write a deleted file */
/* 特殊情况：即使是超级用户(root)也不能读/写一个已被删除的文件 */
// 如果 i 节点有对应的设备，但该 i 节点的连接数等于 0，则返回。
        if (inode->i_dev && !inode->i_nlinks)
                return 0;
// 否则，如果进程的有效用户 id(euid)与 i 节点的用户 id 相同，则取文件宿主的用户访问权限。
        else if (current->euid==inode->i_uid)
                mode >>= 6;
```

```
// 否则，如果进程的有效组 id(egid)与 i 节点的组 id 相同，则取组用户的访问权限。
        else if (current->egid==inode->i_gid)
                mode >>= 3;
// 如果上面所取的的访问权限与屏蔽码相同，或者是超级用户，则返回 1，否则返回 0。
        if (((mode & mask & 0007) == mask) || suser())
                return 1;
        return 0;
}

/*
 * ok, we cannot use strncmp, as the name is not in our data space.
 * Thus we'll have to use match. No big problem. Match also makes
 * some sanity tests.
 *
 * NOTE! unlike strncmp, match returns 1 for success, 0 for failure.
 */
/*
 * ok，我们不能使用 strncmp 字符串比较函数，因为名称不在我们的数据空间(不在内核空间)。
 * 因而我们只能使用 match()。问题不大。match()同样也处理一些完整的测试。
 *
 * 注意！与 strncmp 不同的是 match()成功时返回 1，失败时返回 0。
 */
//// 指定长度字符串比较函数。
// 参数：len - 比较的字符串长度；name - 文件名指针；de - 目录项结构。
// 返回：相同返回 1，不同返回 0。
static int match(int len,const char * name,struct dir_entry * de)
{
        register int same __asm__("ax");

// 如果目录项指针空，或者目录项 i 节点等于 0，或者要比较的字符串长度超过文件名长度，则返回 0。
        if (!de || !de->inode || len > NAME_LEN)
                return 0;
// 如果要比较的长度 len 小于 NAME_LEN，但是目录项中文件名长度超过 len，则返回 0。
        if (len < NAME_LEN && de->name[len])
                return 0;
// 下面嵌入汇编语句，在用户数据空间(fs)执行字符串的比较操作。
// %0 - eax(比较结果 same)；%1 - eax(eax 初值 0)；%2 - esi(名字指针)；%3 - edi(目录项名指针)；
// %4 - ecx(比较的字节长度值 len)。
        __asm__("cld\n\t"                    // 清方向位。
                "fs ; repe ; cmpsb\n\t"      // 用户空间执行循环比较[esi++]和[edi++]操作，
                "setz %%al"                  // 若比较结果一样(z=0)则设置 al=1(same=eax)。
                :"=a" (same)
                :"" (0),"S" ((long) name),"D" ((long) de->name),"c" (len)
                :"cx","di","si");
        return same;                         // 返回比较结果。
}

/*
 *      find_entry()
 *
 * finds an entry in the specified directory with the wanted name. It
 * returns the cache buffer in which the entry was found, and the entry
 * itself (as a parameter - res_dir). It does NOT read the inode of the
```

```
 * entry - you'll have to do that yourself if you want to.
 *
 * This also takes care of the few special cases due to '..'-traversal
 * over a pseudo-root and a mount point.
 */
/*
 *     find_entry()
 * 在指定的目录中寻找一个与名字匹配的目录项。返回一个含有找到目录项的高速
 * 缓冲区以及目录项本身(作为一个参数 - res_dir)。并不读目录项的 i 节点 - 如
 * 果需要的话需自己操作。
 *
 * '..'目录项，操作期间也会对几种特殊情况分别处理 - 比如横越一个伪根目录以
 * 及安装点。
 */
//// 查找指定目录和文件名的目录项。
// 参数：dir - 指定目录 i 节点的指针；name - 文件名；namelen - 文件名长度；
// 返回：高速缓冲区指针；res_dir - 返回的目录项结构指针；
static struct buffer_head * find_entry(struct m_inode ** dir,
        const char * name, int namelen, struct dir_entry ** res_dir)
{
        int entries;
        int block,i;
        struct buffer_head * bh;
        struct dir_entry * de;
        struct super_block * sb;

// 如果定义了 NO_TRUNCATE，则若文件名长度超过最大长度 NAME_LEN，则返回。
#ifdef NO_TRUNCATE
        if (namelen > NAME_LEN)
                return NULL;
//如果没有定义 NO_TRUNCATE，则若文件名长度超过最大长度 NAME_LEN，则截短之。
#else
        if (namelen > NAME_LEN)
                namelen = NAME_LEN;
#endif
// 计算本目录中目录项项数 entries。置空返回目录项结构指针。
        entries = (*dir)->i_size / (sizeof (struct dir_entry));
        *res_dir = NULL;
// 如果文件名长度等于 0，则返回 NULL，退出。
        if (!namelen)
                return NULL;
/* check for '..', as we might have to do some "magic" for it */
/* 检查目录项'..'，因为可能需要对其特别处理 */
        if (namelen==2 && get_fs_byte(name)=='.' && get_fs_byte(name+1)=='.') {
/* '..' in a pseudo-root results in a faked '.' (just change namelen) */
/* 伪根中的'..'如同一个假'.'(只需改变名字长度) */
// 如果当前进程的根节点指针即是指定的目录，则将文件名修改为'.'，
                if ((*dir) == current->root)
                        namelen=1;
// 否则如果该目录的 i 节点号等于 ROOT_INO(1)的话，说明是文件系统根节点。则取文件系统的超级块。

                else if ((*dir)->i_num == ROOT_INO) {
/* '..' over a mount-point results in 'dir' being exchanged for the mounted
```

```
        directory-inode. NOTE! We set mounted, so that we can iput the new dir */
/* 在一个安装点上的'..'将导致目录交换到安装到文件系统的目录 i 节点。
   注意！由于设置了 mounted 标志，因而我们能够取出该新目录 */
                        sb=get_super((*dir)->i_dev);
// 如果被安装到的 i 节点存在，则先释放原 i 节点，然后对被安装到的 i 节点进行处理。
// 让*dir 指向该被安装到的 i 节点；该 i 节点的引用数加 1。
                        if (sb->s_imount) {
                                iput(*dir);
                                (*dir)=sb->s_imount;
                                (*dir)->i_count++;
                        }
                }
        }
// 如果该 i 节点所指向的第一个直接磁盘块号为 0，则返回 NULL，退出。
        if (!(block = (*dir)->i_zone[0]))
                return NULL;
// 从节点所在设备读取指定的目录项数据块，如果不成功，则返回 NULL，退出。
        if (!(bh = bread((*dir)->i_dev,block)))
                return NULL;
// 在目录项数据块中搜索匹配指定文件名的目录项，首先让 de 指向数据块，并在不超过目录中目录项数
// 的条件下，循环执行搜索。
        i = 0;
        de = (struct dir_entry *) bh->b_data;
        while (i < entries) {
// 如果当前目录项数据块已经搜索完，还没有找到匹配的目录项，则释放当前目录项数据块。
                if ((char *)de >= BLOCK_SIZE+bh->b_data) {
                        brelse(bh);
                        bh = NULL;
// 在读入下一目录项数据块。若这块为空，则只要还没有搜索完目录中的所有目录项，就跳过该块，
// 继续读下一目录项数据块。若该块不空，就让 de 指向该目录项数据块，继续搜索。
                        if (!(block = bmap(*dir,i/DIR_ENTRIES_PER_BLOCK)) ||
                            !(bh = bread((*dir)->i_dev,block))) {
                                i += DIR_ENTRIES_PER_BLOCK;
                                continue;
                        }
                        de = (struct dir_entry *) bh->b_data;
                }
// 如果找到匹配的目录项的话，则返回该目录项结构指针和该目录项数据块指针，退出。
                if (match(namelen,name,de)) {
                        *res_dir = de;
                        return bh;
                }
// 否则继续在目录项数据块中比较下一个目录项。
                de++;
                i++;
        }
// 若指定目录中的所有目录项都搜索完还没有找到相应的目录项，则释放目录项数据块，返回 NULL。
        brelse(bh);
        return NULL;
}

/*
```

```
 *      add_entry()
 *
 * adds a file entry to the specified directory, using the same
 * semantics as find_entry(). It returns NULL if it failed.
 *
 * NOTE!! The inode part of 'de' is left at 0 - which means you
 * may not sleep between calling this and putting something into
 * the entry, as someone else might have used it while you slept.
 */
/*
 *      add_entry()
 * 使用与 find_entry()同样的方法，往指定目录中添加一文件目录项。
 * 如果失败则返回 NULL。
 *
 * 注意！！'de'(指定目录项结构指针)的 i 节点部分被设置为 0 - 这表示
 * 在调用该函数和往目录项中添加信息之间不能睡眠，因为若睡眠那么其它
 * 人(进程)可能会已经使用了该目录项。
 */
//// 根据指定的目录和文件名添加目录项。
// 参数：dir - 指定目录的 i 节点；name - 文件名；namelen - 文件名长度；
// 返回：高速缓冲区指针；res_dir - 返回的目录项结构指针；
static struct buffer_head * add_entry(struct m_inode * dir,
        const char * name, int namelen, struct dir_entry ** res_dir)
{
        int block,i;
        struct buffer_head * bh;
        struct dir_entry * de;

        *res_dir = NULL;
// 如果定义了 NO_TRUNCATE，则若文件名长度超过最大长度 NAME_LEN，则返回。
#ifdef NO_TRUNCATE
        if (namelen > NAME_LEN)
                return NULL;
//如果没有定义 NO_TRUNCATE，则若文件名长度超过最大长度 NAME_LEN，则截短之。
#else
        if (namelen > NAME_LEN)
                namelen = NAME_LEN;
#endif
// 如果文件名长度等于 0，则返回 NULL，退出。
        if (!namelen)
                return NULL;
// 如果该目录 i 节点所指向的第一个直接磁盘块号为 0，则返回 NULL 退出。
        if (!(block = dir->i_zone[0]))
                return NULL;
// 如果读取该磁盘块失败，则返回 NULL 并退出。
        if (!(bh = bread(dir->i_dev,block)))
                return NULL;
// 在目录项数据块中循环查找最后未使用的目录项。首先让目录项结构指针 de 指向高速缓冲的数据块
// 开始处，也即第一个目录项。
        i = 0;
        de = (struct dir_entry *) bh->b_data;
        while (1) {
// 如果当前判别的目录项已经超出当前数据块，则释放该数据块，重新申请一块磁盘块 block。如果
```

```
// 申请失败，则返回 NULL，退出。
                if ((char *)de >= BLOCK_SIZE+bh->b_data) {
                        brelse(bh);
                        bh = NULL;
                        block = create_block(dir,i/DIR_ENTRIES_PER_BLOCK);
                        if (!block)
                                return NULL;
// 如果读取磁盘块返回的指针为空，则跳过该块继续。
                        if (!(bh = bread(dir->i_dev,block))) {
                                i += DIR_ENTRIES_PER_BLOCK;
                                continue;
                        }
// 否则，让目录项结构指针 de 志向该块的高速缓冲数据块开始处。
                        de = (struct dir_entry *) bh->b_data;
                }
// 如果当前所操作的目录项序号 i*目录结构大小已经超过了该目录所指出的大小 i_size，则说明该第 i
// 个目录项还未使用，我们可以使用它。于是对该目录项进行设置(置该目录项的 i 节点指针为空)。并
// 更新该目录的长度值(加上一个目录项的长度，设置目录的 i 节点已修改标志，再更新该目录的改变时
// 间为当前时间。
                if (i*sizeof(struct dir_entry) >= dir->i_size) {
                        de->inode=0;
                        dir->i_size = (i+1)*sizeof(struct dir_entry);
                        dir->i_dirt = 1;
                        dir->i_ctime = CURRENT_TIME;
                }
// 若该目录项的 i 节点为空，则表示找到一个还未使用的目录项。于是更新目录的修改时间为当前时间。
// 并从用户数据区复制文件名到该目录项的文件名字段，置相应的高速缓冲块已修改标志。返回该目录
// 项的指针以及该高速缓冲区的指针，退出。
                if (!de->inode) {
                        dir->i_mtime = CURRENT_TIME;
                        for (i=0; i < NAME_LEN ; i++)
                                de->name[i]=(i<namelen)?get_fs_byte(name+i):0;
                        bh->b_dirt = 1;
                        *res_dir = de;
                        return bh;
                }
// 如果该目录项已经被使用，则继续检测下一个目录项。
                de++;
                i++;
        }
// 执行不到这里。也许 Linus 在写这段代码时是先复制了上面 find_entry()的代码，而后修改的☺。
        brelse(bh);
        return NULL;
}

/*
 *      get_dir()
 *
 * Getdir traverses the pathname until it hits the topmost directory.
 * It returns NULL on failure.
 */
/*
```

```
 *      get_dir()
 * 该函数根据给出的路径名进行搜索，直到达到最顶端的目录。
 * 如果失败则返回 NULL。
 */
//// 搜寻指定路径名的目录。
// 参数：pathname - 路径名。
// 返回：目录的 i 节点指针。失败时返回 NULL。
static struct m_inode * get_dir(const char * pathname)
{
        char c;
        const char * thisname;
        struct m_inode * inode;
        struct buffer_head * bh;
        int namelen,inr,idev;
        struct dir_entry * de;

// 如果进程没有设定根 i 节点，或者该进程根 i 节点的引用为 0，则系统出错，死机。
        if (!current->root || !current->root->i_count)
                panic("No root inode");
// 如果进程的当前工作目录指针为空，或者该当前目录 i 节点的引用计数为 0，也是系统有问题，死机。
        if (!current->pwd || !current->pwd->i_count)
                panic("No cwd inode");
// 如果用户指定的路径名的第 1 个字符是'/'，则说明路径名是绝对路径名。则从根 i 节点开始操作。
        if ((c=get_fs_byte(pathname))=='/') {
                inode = current->root;
                pathname++;
// 否则若第一个字符是其它字符，则表示给定的是相对路径名。应从进程的当前工作目录开始操作。
// 则取进程当前工作目录的 i 节点。
        } else if (c)
                inode = current->pwd;
// 否则表示路径名为空，出错。返回 NULL，退出。
        else
                return NULL;    /* empty name is bad */   /* 空的路径名是错误的 */
// 将取得的 i 节点引用计数增 1。
        inode->i_count++;
        while (1) {
// 若该 i 节点不是目录节点，或者没有可进入的访问许可，则释放该 i 节点，返回 NULL，退出。
                thisname = pathname;
                if (!S_ISDIR(inode->i_mode) || !permission(inode,MAY_EXEC)) {
                        iput(inode);
                        return NULL;
                }
// 从路径名开始起搜索检测字符，直到字符已是结尾符(NULL)或者是'/'，此时 namelen 正好是当前处理
// 目录名的长度。如果最后也是一个目录名，但其后没有加'/'，则不会返回该最后目录的 i 节点！
// 比如：/var/log/httpd，将只返回 log/目录的 i 节点。
                for(namelen=0;(c=get_fs_byte(pathname++))&&(c!='/');namelen++)
                        /* nothing */ ;
// 若字符是结尾符 NULL，则表明已经到达指定目录，则返回该 i 节点指针，退出。
                if (!c)
                        return inode;
// 调用查找指定目录和文件名的目录项函数，在当前处理目录中寻找子目录项。如果没有找到，则释放
// 该 i 节点，并返回 NULL，退出。
```

```
                if (!(bh = find_entry(&inode,thisname,namelen,&de))) {
                        iput(inode);
                        return NULL;
                }
// 取该子目录项的 i 节点号 inr 和设备号 idev，释放包含该目录项的高速缓冲块和该 i 节点。
                inr = de->inode;
                idev = inode->i_dev;
                brelse(bh);
                iput(inode);
// 取节点号 inr 的 i 节点信息，若失败，则返回 NULL，退出。否则继续以该子目录的 i 节点进行操作。
                if (!(inode = iget(idev,inr)))
                        return NULL;
        }
}

/*
 *      dir_namei()
 *
 * dir_namei() returns the inode of the directory of the
 * specified name, and the name within that directory.
 */
/*
 *      dir_namei()
 * dir_namei()函数返回指定目录名的 i 节点指针，以及在最顶层目录的名称。
 */
// 参数：pathname - 目录路径名；namelen - 路径名长度。
// 返回：指定目录名最顶层目录的 i 节点指针和最顶层目录名及其长度。
static struct m_inode * dir_namei(const char * pathname,
        int * namelen, const char ** name)
{
        char c;
        const char * basename;
        struct m_inode * dir;

// 取指定路径名最顶层目录的 i 节点，若出错则返回 NULL，退出。
        if (!(dir = get_dir(pathname)))
                return NULL;
// 对路径名 pathname 进行搜索检测，查处最后一个'/'后面的名字字符串，计算其长度，并返回最顶
// 层目录的 i 节点指针。
        basename = pathname;
        while (c=get_fs_byte(pathname++))
                if (c=='/')
                        basename=pathname;
        *namelen = pathname-basename-1;
        *name = basename;
        return dir;
}

/*
 *      namei()
 *
 * is used by most simple commands to get the inode of a specified name.
 * Open, link etc use their own routines, but this is enough for things
```

```
 * like 'chmod' etc.
 */
/*
 *      namei()
 * 该函数被许多简单的命令用于取得指定路径名称的 i 节点。open、link 等则使用它们
 * 自己的相应函数，但对于象修改模式'chmod'等这样的命令，该函数已足够用了。
 */
//// 取指定路径名的 i 节点。
// 参数：pathname - 路径名。
// 返回：对应的 i 节点。
struct m_inode * namei(const char * pathname)
{
        const char * basename;
        int inr,dev,namelen;
        struct m_inode * dir;
        struct buffer_head * bh;
        struct dir_entry * de;

// 首先查找指定路径的最顶层目录的目录名及其 i 节点，若不存在，则返回 NULL，退出。
        if (!(dir = dir_namei(pathname,&namelen,&basename)))
                return NULL;
// 如果返回的最顶层名字的长度是 0，则表示该路径名以一个目录名为最后一项。
        if (!namelen)                   /* special case: '/usr/' etc */
                return dir;             /* 对应于'/usr/'等情况 */
// 在返回的顶层目录中寻找指定文件名的目录项的 i 节点。因为如果最后也是一个目录名，但其后没
// 有加'/'，则不会返回该最后目录的 i 节点！比如：/var/log/httpd，将只返回 log/目录的 i 节点。
// 因此 dir_namei()将不以'/'结束的最后一个名字当作一个文件名来看待。因此这里需要单独对这种
// 情况使用寻找目录项 i 节点函数 find_entry()进行处理。
        bh = find_entry(&dir,basename,namelen,&de);
        if (!bh) {
                iput(dir);
                return NULL;
        }
// 取该目录项的 i 节点号和目录的设备号，并释放包含该目录项的高速缓冲区以及目录 i 节点。
        inr = de->inode;
        dev = dir->i_dev;
        brelse(bh);
        iput(dir);
// 取对应节号的 i 节点，修改其被访问时间为当前时间，并置已修改标志。最后返回该 i 节点指针。
        dir=iget(dev,inr);
        if (dir) {
                dir->i_atime=CURRENT_TIME;
                dir->i_dirt=1;
        }
        return dir;
}

/*
 *      open_namei()
 *
 * namei for open - this is in fact almost the whole open-routine.
 */
/*
```

```
 *      open_namei()
 * open()所使用的 namei 函数 - 这其实几乎是完整的打开文件程序。
 */
//// 文件打开 namei 函数。
// 参数：pathname - 文件路径名；flag - 文件打开标志；mode - 文件访问许可属性；
// 返回：成功返回 0，否则返回出错码；res_inode - 返回的对应文件路径名的的 i 节点指针。
int open_namei(const char * pathname, int flag, int mode,
        struct m_inode ** res_inode)
{
        const char * basename;
        int inr,dev,namelen;
        struct m_inode * dir, *inode;
        struct buffer_head * bh;
        struct dir_entry * de;

// 如果文件访问许可模式标志是只读(0)，但文件截 0 标志 O_TRUNC 却置位了，则改为只写标志。
        if ((flag & O_TRUNC) && !(flag & O_ACCMODE))
                flag |= O_WRONLY;
// 使用进程的文件访问许可屏蔽码，屏蔽掉给定模式中的相应位，并添上普通文件标志。
        mode &= 0777 & ~current->umask;
        mode |= I_REGULAR;
// 根据路径名寻找到对应的 i 节点，以及最顶端文件名及其长度。
        if (!(dir = dir_namei(pathname,&namelen,&basename)))
                return -ENOENT;
// 如果最顶端文件名长度为 0(例如'/usr/'这种路径名的情况)，那么若打开操作不是创建、截 0，
// 则表示打开一个目录名，直接返回该目录的 i 节点，并退出。
        if (!namelen) {                 /* special case: '/usr/' etc */
                if (!(flag & (O_ACCMODE|O_CREAT|O_TRUNC))) {
                        *res_inode=dir;
                        return 0;
                }
// 否则释放该 i 节点，返回出错码。
                iput(dir);
                return -EISDIR;
        }
// 在 dir 节点对应的目录中取文件名对应的目录项结构 de 和该目录项所在的高速缓冲区。
        bh = find_entry(&dir,basename,namelen,&de);
// 如果该高速缓冲指针为 NULL，则表示没有找到对应文件名的目录项，因此只可能是创建文件操作。
        if (!bh) {
// 如果不是创建文件，则释放该目录的 i 节点，返回出错号退出。
                if (!(flag & O_CREAT)) {
                        iput(dir);
                        return -ENOENT;
                }
// 如果用户在该目录没有写的权力，则释放该目录的 i 节点，返回出错号退出。
                if (!permission(dir,MAY_WRITE)) {
                        iput(dir);
                        return -EACCES;
                }
// 在目录节点对应的设备上申请一个新 i 节点，若失败，则释放目录的 i 节点，并返回没有空间出错码。
                inode = new_inode(dir->i_dev);
                if (!inode) {
                        iput(dir);
```

```
                        return -ENOSPC;
                }
// 否则使用该新 i 节点，对其进行初始设置：置节点的用户 id；对应节点访问模式；置已修改标志。
                inode->i_uid = current->euid;
                inode->i_mode = mode;
                inode->i_dirt = 1;
// 然后在指定目录 dir 中添加一新目录项。
                bh = add_entry(dir,basename,namelen,&de);
// 如果返回的应该含有新目录项的高速缓冲区指针为 NULL，则表示添加目录项操作失败。于是将该
// 新 i 节点的引用连接计数减 1；并释放该 i 节点与目录的 i 节点，返回出错码，退出。
                if (!bh) {
                        inode->i_nlinks--;
                        iput(inode);
                        iput(dir);
                        return -ENOSPC;
                }
// 初始设置该新目录项：置 i 节点号为新申请到的 i 节点的号码；并置高速缓冲区已修改标志。然后
// 释放该高速缓冲区，释放目录的 i 节点。返回新目录项的 i 节点指针，退出。
                de->inode = inode->i_num;
                bh->b_dirt = 1;
                brelse(bh);
                iput(dir);
                *res_inode = inode;
                return 0;
        }
// 若上面在目录中取文件名对应的目录项结构操作成功(也即 bh 不为 NULL)，取出该目录项的 i 节点号
// 和其所在的设备号，并释放该高速缓冲区以及目录的 i 节点。
        inr = de->inode;
        dev = dir->i_dev;
        brelse(bh);
        iput(dir);
// 如果独占使用标志 O_EXCL 置位，则返回文件已存在出错码，退出。
        if (flag & O_EXCL)
                return -EEXIST;
// 如果取该目录项对应 i 节点的操作失败，则返回访问出错码，退出。
        if (!(inode=iget(dev,inr)))
                return -EACCES;
// 若该 i 节点是一个目录的节点并且访问模式是只读或只写，或者没有访问的许可权限，则释放该 i
// 节点，返回访问权限出错码，退出。
        if ((S_ISDIR(inode->i_mode) && (flag & O_ACCMODE)) ||
            !permission(inode,ACC_MODE(flag))) {
                iput(inode);
                return -EPERM;
        }
// 更新该 i 节点的访问时间字段为当前时间。
        inode->i_atime = CURRENT_TIME;
// 如果设立了截 0 标志，则将该 i 节点的文件长度截为 0。
        if (flag & O_TRUNC)
                truncate(inode);
// 最后返回该目录项 i 节点的指针，并返回 0（成功）。
        *res_inode = inode;
        return 0;
}
```

```
//// 系统调用函数 - 创建一个特殊文件或普通文件节点(node)。
// 创建名称为 filename，由 mode 和 dev 指定的文件系统节点(普通文件、设备特殊文件或命名管道)。
// 参数：filename - 路径名；mode - 指定使用许可以及所创建节点的类型；dev - 设备号。
// 返回：成功则返回 0，否则返回出错码。
int sys_mknod(const char * filename, int mode, int dev)
{
        const char * basename;
        int namelen;
        struct m_inode * dir, * inode;
        struct buffer_head * bh;
        struct dir_entry * de;

// 如果不是超级用户，则返回访问许可出错码。
        if (!suser())
                return -EPERM;
// 如果找不到对应路径名目录的 i 节点，则返回出错码。
        if (!(dir = dir_namei(filename,&namelen,&basename)))
                return -ENOENT;
// 如果最顶端的文件名长度为 0，则说明给出的路径名最后没有指定文件名，释放该目录 i 节点，返回
// 出错码，退出。
        if (!namelen) {
                iput(dir);
                return -ENOENT;
        }
// 如果在该目录中没有写的权限，则释放该目录的 i 节点，返回访问许可出错码，退出。
        if (!permission(dir,MAY_WRITE)) {
                iput(dir);
                return -EPERM;
        }
// 如果对应路径名上最后的文件名的目录项已经存在，则释放包含该目录项的高速缓冲区，释放目录
// 的 i 节点，返回文件已经存在出错码，退出。
        bh = find_entry(&dir,basename,namelen,&de);
        if (bh) {
                brelse(bh);
                iput(dir);
                return -EEXIST;
        }
// 申请一个新的 i 节点，如果不成功，则释放目录的 i 节点，返回无空间出错码，退出。
        inode = new_inode(dir->i_dev);
        if (!inode) {
                iput(dir);
                return -ENOSPC;
        }
// 设置该 i 节点的属性模式。如果要创建的是块设备文件或者是字符设备文件，则令 i 节点的直接块
// 指针 0 等于设备号。
        inode->i_mode = mode;
        if (S_ISBLK(mode) || S_ISCHR(mode))
                inode->i_zone[0] = dev;
// 设置该 i 节点的修改时间、访问时间为当前时间。
        inode->i_mtime = inode->i_atime = CURRENT_TIME;
        inode->i_dirt = 1;
// 在目录中新添加一个目录项，如果失败(包含该目录项的高速缓冲区指针为 NULL)，则释放目录的
```

```
// i 节点；所申请的 i 节点引用连接计数复位，并释放该 i 节点。返回出错码，退出。
        bh = add_entry(dir,basename,namelen,&de);
        if (!bh) {
                iput(dir);
                inode->i_nlinks=0;
                iput(inode);
                return -ENOSPC;
        }
// 令该目录项的 i 节点字段等于新 i 节点号，置高速缓冲区已修改标志，释放目录和新的 i 节点，释放
// 高速缓冲区，最后返回 0(成功)。
        de->inode = inode->i_num;
        bh->b_dirt = 1;
        iput(dir);
        iput(inode);
        brelse(bh);
        return 0;
}

//// 系统调用函数 - 创建目录。
// 参数：pathname - 路径名；mode - 目录使用的权限属性。
// 返回：成功则返回 0，否则返回出错码。
int sys_mkdir(const char * pathname, int mode)
{
        const char * basename;
        int namelen;
        struct m_inode * dir, * inode;
        struct buffer_head * bh, *dir_block;
        struct dir_entry * de;

// 如果不是超级用户，则返回访问许可出错码。
        if (!suser())
                return -EPERM;
// 如果找不到对应路径名目录的 i 节点，则返回出错码。
        if (!(dir = dir_namei(pathname,&namelen,&basename)))
                return -ENOENT;
// 如果最顶端的文件名长度为 0，则说明给出的路径名最后没有指定文件名，释放该目录 i 节点，返回
// 出错码，退出。
        if (!namelen) {
                iput(dir);
                return -ENOENT;
        }
// 如果在该目录中没有写的权限，则释放该目录的 i 节点，返回访问许可出错码，退出。
        if (!permission(dir,MAY_WRITE)) {
                iput(dir);
                return -EPERM;
        }
// 如果对应路径名上最后的文件名的目录项已经存在，则释放包含该目录项的高速缓冲区，释放目录
// 的 i 节点，返回文件已经存在出错码，退出。
        bh = find_entry(&dir,basename,namelen,&de);
        if (bh) {
                brelse(bh);
                iput(dir);
                return -EEXIST;
```

```
        }
// 申请一个新的 i 节点，如果不成功，则释放目录的 i 节点，返回无空间出错码，退出。
        inode = new_inode(dir->i_dev);
        if (!inode) {
                iput(dir);
                return -ENOSPC;
        }
// 置该新 i 节点对应的文件长度为 32(一个目录项的大小)，置节点已修改标志，以及节点的修改时间
// 和访问时间。
        inode->i_size = 32;
        inode->i_dirt = 1;
        inode->i_mtime = inode->i_atime = CURRENT_TIME;
// 为该 i 节点申请一磁盘块，并令节点第一个直接块指针等于该块号。如果申请失败，则释放对应目录
// 的 i 节点；复位新申请的 i 节点连接计数；释放该新的 i 节点，返回没有空间出错码，退出。
        if (!(inode->i_zone[0]=new_block(inode->i_dev))) {
                iput(dir);
                inode->i_nlinks--;
                iput(inode);
                return -ENOSPC;
        }
// 置该新的 i 节点已修改标志。
        inode->i_dirt = 1;
// 读新申请的磁盘块。若出错，则释放对应目录的 i 节点；释放申请的磁盘块；复位新申请的 i 节点
// 连接计数；释放该新的 i 节点，返回没有空间出错码，退出。
        if (!(dir_block=bread(inode->i_dev,inode->i_zone[0]))) {
                iput(dir);
                free_block(inode->i_dev,inode->i_zone[0]);
                inode->i_nlinks--;
                iput(inode);
                return -ERROR;
        }
// 令 de 指向目录项数据块，置该目录项的 i 节点号字段等于新申请的 i 节点号，名字字段等于"."。
        de = (struct dir_entry *) dir_block->b_data;
        de->inode=inode->i_num;
        strcpy(de->name,".");
// 然后 de 指向下一个目录项结构，该结构用于存放上级目录的节点号和名字".."。
        de++;
        de->inode = dir->i_num;
        strcpy(de->name,"..");
        inode->i_nlinks = 2;
// 然后设置该高速缓冲区已修改标志，并释放该缓冲区。
        dir_block->b_dirt = 1;
        brelse(dir_block);
// 初始化设置新 i 节点的模式字段，并置该 i 节点已修改标志。
        inode->i_mode = I_DIRECTORY | (mode & 0777 & ~current->umask);
        inode->i_dirt = 1;
// 在目录中新添加一个目录项，如果失败(包含该目录项的高速缓冲区指针为 NULL)，则释放目录的
// i 节点；所申请的 i 节点引用连接计数复位，并释放该 i 节点。返回出错码，退出。
        bh = add_entry(dir,basename,namelen,&de);
        if (!bh) {
                iput(dir);
                free_block(inode->i_dev,inode->i_zone[0]);
                inode->i_nlinks=0;
```

```
                iput(inode);
                return -ENOSPC;
        }
// 令该目录项的 i 节点字段等于新 i 节点号，置高速缓冲区已修改标志，释放目录和新的 i 节点，释放
// 高速缓冲区，最后返回 0(成功)。
        de->inode = inode->i_num;
        bh->b_dirt = 1;
        dir->i_nlinks++;
        dir->i_dirt = 1;
        iput(dir);
        iput(inode);
        brelse(bh);
        return 0;
}

/*
 * routine to check that the specified directory is empty (for rmdir)
 */
/*
 * 用于检查指定的目录是否为空的子程序(用于 rmdir 系统调用函数)。
 */
//// 检查指定目录是否是空的。
// 参数：inode - 指定目录的 i 节点指针。
// 返回：0 - 是空的；1 - 不空。
static int empty_dir(struct m_inode * inode)
{
        int nr,block;
        int len;
        struct buffer_head * bh;
        struct dir_entry * de;

// 计算指定目录中现有目录项的个数(应该起码有 2 个，即"."和".."两个文件目录项)。
        len = inode->i_size / sizeof (struct dir_entry);
// 如果目录项个数少于 2 个或者该目录 i 节点的第 1 个直接块没有指向任何磁盘块号，或者相应磁盘
// 块读不出，则显示警告信息“设备 dev 上目录错”，返回 0(失败)。
        if (len<2 || !inode->i_zone[0] ||
            !(bh=bread(inode->i_dev,inode->i_zone[0]))) {
                printk("warning - bad directory on dev %04x\n",inode->i_dev);
                return 0;
        }
// 让 de 指向含有读出磁盘块数据的高速缓冲区中第 1 项目录项。
        de = (struct dir_entry *) bh->b_data;
// 如果第 1 个目录项的 i 节点号字段值不等于该目录的 i 节点号，或者第 2 个目录项的 i 节点号字段
// 为零，或者两个目录项的名字字段不分别等于"."和".."，则显示出错警告信息“设备 dev 上目录错”
// 并返回 0。
        if (de[0].inode != inode->i_num || !de[1].inode ||
            strcmp(".",de[0].name) || strcmp("..",de[1].name)) {
                printk("warning - bad directory on dev %04x\n",inode->i_dev);
                return 0;
        }
// 令 nr 等于目录项序号；de 指向第三个目录项。
        nr = 2;
        de += 2;
```

```
// 循环检测该目录中所有的目录项(len-2 个)，看有没有目录项的 i 节点号字段不为 0(被使用)。
        while (nr<len) {
// 如果该块磁盘块中的目录项已经检测完，则释放该磁盘块的高速缓冲区，读取下一块含有目录项的
// 磁盘块。若相应块没有使用(或已经不用，如文件已经删除等)，则继续读下一块，若读不出，则出
// 错，返回 0。否则让 de 指向读出块的首个目录项。
                if ((void *) de >= (void *) (bh->b_data+BLOCK_SIZE)) {
                        brelse(bh);
                        block=bmap(inode,nr/DIR_ENTRIES_PER_BLOCK);
                        if (!block) {
                                nr += DIR_ENTRIES_PER_BLOCK;
                                continue;
                        }
                        if (!(bh=bread(inode->i_dev,block)))
                                return 0;
                        de = (struct dir_entry *) bh->b_data;
                }
// 如果该目录项的 i 节点号字段不等于 0，则表示该目录项目前正被使用，则释放该高速缓冲区，
// 返回 0，退出。
                if (de->inode) {
                        brelse(bh);
                        return 0;
                }
// 否则，若还没有查询完该目录中的所有目录项，则继续检测。
                de++;
                nr++;
        }
// 到这里说明该目录中没有找到已用的目录项(当然除了头两个以外)，则返回缓冲区，返回 1。
        brelse(bh);
        return 1;
}

//// 系统调用函数 - 删除指定名称的目录。
// 参数： name - 目录名(路径名)。
// 返回：返回 0 表示成功，否则返回出错号。
int sys_rmdir(const char * name)
{
        const char * basename;
        int namelen;
        struct m_inode * dir, * inode;
        struct buffer_head * bh;
        struct dir_entry * de;

// 如果不是超级用户，则返回访问许可出错码。
        if (!suser())
                return -EPERM;
// 如果找不到对应路径名目录的 i 节点，则返回出错码。
        if (!(dir = dir_namei(name,&namelen,&basename)))
                return -ENOENT;
// 如果最顶端的文件名长度为 0，则说明给出的路径名最后没有指定文件名，释放该目录 i 节点，返回
// 出错码，退出。
        if (!namelen) {
                iput(dir);
                return -ENOENT;
```

```
        }
// 如果在该目录中没有写的权限，则释放该目录的 i 节点，返回访问许可出错码，退出。
        if (!permission(dir,MAY_WRITE)) {
                iput(dir);
                return -EPERM;
        }
// 如果对应路径名上最后的文件名的目录项不存在，则释放包含该目录项的高速缓冲区，释放目录
// 的 i 节点，返回文件已经存在出错码，退出。否则 dir 是包含要被删除目录名的目录 i 节点，de
// 是要被删除目录的目录项结构。
        bh = find_entry(&dir,basename,namelen,&de);
        if (!bh) {
                iput(dir);
                return -ENOENT;
        }
// 取该目录项指明的 i 节点。若出错则释放目录的 i 节点，并释放含有目录项的高速缓冲区，返回
// 出错号。
        if (!(inode = iget(dir->i_dev, de->inode))) {
                iput(dir);
                brelse(bh);
                return -EPERM;
        }
// 若该目录设置了受限删除标志并且进程的有效用户 id 不等于该 i 节点的用户 id，则表示没有权限删
// 除该目录，于是释放包含要删除目录名的目录 i 节点和该要删除目录的 i 节点，释放高速缓冲区，返
// 回出错码。
        if ((dir->i_mode & S_ISVTX) && current->euid &&
            inode->i_uid != current->euid) {
                iput(dir);
                iput(inode);
                brelse(bh);
                return -EPERM;
        }
// 如果要被删除的目录项的 i 节点的设备号不等于包含该目录项的目录的设备号，或者该被删除目录的
// 引用连接计数大于 1（表示有符号连接等），则不能删除该目录，于是释放包含要删除目录名的目录
// i 节点和该要删除目录的 i 节点，释放高速缓冲区，返回出错码。
        if (inode->i_dev != dir->i_dev || inode->i_count>1) {
                iput(dir);
                iput(inode);
                brelse(bh);
                return -EPERM;
        }
// 如果要被删除目录的目录项 i 节点的节点号等于包含该需删除目录的 i 节点号，则表示试图删除"."
// 目录。于是释放包含要删除目录名的目录 i 节点和该要删除目录的 i 节点，释放高速缓冲区，返回
// 出错码。
        if (inode == dir) {     /* we may not delete ".", but "../dir" is ok */
                iput(inode);    /* 我们不可以删除"."，但可以删除"../dir" */
                iput(dir);
                brelse(bh);
                return -EPERM;
        }
// 若要被删除的目录的 i 节点的属性表明这不是一个目录，则释放包含要删除目录名的目录 i 节点和
// 该要删除目录的 i 节点，释放高速缓冲区，返回出错码。
        if (!S_ISDIR(inode->i_mode)) {
                iput(inode);
```

```
                iput(dir);
                brelse(bh);
                return -ENOTDIR;
        }
// 若该需被删除的目录不空，则释放包含要删除目录名的目录 i 节点和该要删除目录的 i 节点，释放
// 高速缓冲区，返回出错码。
        if (!empty_dir(inode)) {
                iput(inode);
                iput(dir);
                brelse(bh);
                return -ENOTEMPTY;
        }
// 若该需被删除目录的 i 节点的连接数不等于 2，则显示警告信息。
        if (inode->i_nlinks != 2)
                printk("empty directory has nlink!=2 (%d)",inode->i_nlinks);
// 置该需被删除目录的目录项的 i 节点号字段为 0，表示该目录项不再使用，并置含有该目录项的高速
// 缓冲区已修改标志，并释放该缓冲区。
        de->inode = 0;
        bh->b_dirt = 1;
        brelse(bh);
// 置被删除目录的 i 节点的连接数为 0，并置 i 节点已修改标志。
        inode->i_nlinks=0;
        inode->i_dirt=1;
// 将包含被删除目录名的目录的 i 节点引用计数减 1，修改其改变时间和修改时间为当前时间，并置
// 该节点已修改标志。
        dir->i_nlinks--;
        dir->i_ctime = dir->i_mtime = CURRENT_TIME;
        dir->i_dirt=1;
// 最后释放包含要删除目录名的目录 i 节点和该要删除目录的 i 节点，返回 0(成功)。
        iput(dir);
        iput(inode);
        return 0;
}

//// 系统调用函数 - 删除文件名以及可能也删除其相关的文件。
// 从文件系统删除一个名字。如果是一个文件的最后一个连接，并且没有进程正打开该文件，则该文件
// 也将被删除，并释放所占用的设备空间。
// 参数：name - 文件名。
// 返回：成功则返回 0，否则返回出错号。
int sys_unlink(const char * name)
{
        const char * basename;
        int namelen;
        struct m_inode * dir, * inode;
        struct buffer_head * bh;
        struct dir_entry * de;

// 如果找不到对应路径名目录的 i 节点，则返回出错码。
        if (!(dir = dir_namei(name,&namelen,&basename)))
                return -ENOENT;
// 如果最顶端的文件名长度为 0，则说明给出的路径名最后没有指定文件名，释放该目录 i 节点，返回
// 出错码，退出。
        if (!namelen) {
```

```
                iput(dir);
                return -ENOENT;
        }
// 如果在该目录中没有写的权限，则释放该目录的 i 节点，返回访问许可出错码，退出。
        if (!permission(dir,MAY_WRITE)) {
                iput(dir);
                return -EPERM;
        }
// 如果对应路径名上最后的文件名的目录项不存在，则释放包含该目录项的高速缓冲区，释放目录
// 的 i 节点，返回文件已经存在出错码，退出。否则 dir 是包含要被删除目录名的目录 i 节点，de
// 是要被删除目录的目录项结构。
        bh = find_entry(&dir,basename,namelen,&de);
        if (!bh) {
                iput(dir);
                return -ENOENT;
        }
// 取该目录项指明的 i 节点。若出错则释放目录的 i 节点，并释放含有目录项的高速缓冲区，返回
// 出错号。
        if (!(inode = iget(dir->i_dev, de->inode))) {
                iput(dir);
                brelse(bh);
                return -ENOENT;
        }
// 如果该目录设置了受限删除标志并且用户不是超级用户，并且进程的有效用户 id 不等于被删除文件
// 名 i 节点的用户 id，并且进程的有效用户 id 也不等于目录 i 节点的用户 id，则没有权限删除该文件
// 名。则释放该目录 i 节点和该文件名目录项的 i 节点，释放包含该目录项的缓冲区，返回出错号。
        if ((dir->i_mode & S_ISVTX) && !suser() &&
            current->euid != inode->i_uid &&
            current->euid != dir->i_uid) {
                iput(dir);
                iput(inode);
                brelse(bh);
                return -EPERM;
        }
// 如果该指定文件名是一个目录，则也不能删除，释放该目录 i 节点和该文件名目录项的 i 节点，释放
// 包含该目录项的缓冲区，返回出错号。
        if (S_ISDIR(inode->i_mode)) {
                iput(inode);
                iput(dir);
                brelse(bh);
                return -EPERM;
        }
// 如果该 i 节点的连接数已经为 0，则显示警告信息，修正其为 1。
        if (!inode->i_nlinks) {
                printk("Deleting nonexistent file (%04x:%d), %d\n",
                        inode->i_dev,inode->i_num,inode->i_nlinks);
                inode->i_nlinks=1;
        }
// 将该文件名的目录项中的 i 节点号字段置为 0，表示释放该目录项，并设置包含该目录项的缓冲区
// 已修改标志，释放该高速缓冲区。
        de->inode = 0;
        bh->b_dirt = 1;
        brelse(bh);
```

```
// 该 i 节点的连接数减 1，置已修改标志，更新改变时间为当前时间。最后释放该 i 节点和目录的 i 节
// 点，返回 0(成功)。
        inode->i_nlinks--;
        inode->i_dirt = 1;
        inode->i_ctime = CURRENT_TIME;
        iput(inode);
        iput(dir);
        return 0;
}

//// 系统调用函数 - 为文件建立一个文件名。
// 为一个已经存在的文件创建一个新连接(也称为硬连接 - hard link)。
// 参数：oldname - 原路径名；newname - 新的路径名。
// 返回：若成功则返回 0，否则返回出错号。
int sys_link(const char * oldname, const char * newname)
{
        struct dir_entry * de;
        struct m_inode * oldinode, * dir;
        struct buffer_head * bh;
        const char * basename;
        int namelen;

// 取原文件路径名对应的 i 节点 oldinode。如果为 0，则表示出错，返回出错号。
        oldinode=namei(oldname);
        if (!oldinode)
                return -ENOENT;
// 如果原路径名对应的是一个目录名，则释放该 i 节点，返回出错号。
        if (S_ISDIR(oldinode->i_mode)) {
                iput(oldinode);
                return -EPERM;
        }
// 查找新路径名的最顶层目录的 i 节点，并返回最后的文件名及其长度。如果目录的 i 节点没有找到，
// 则释放原路径名的 i 节点，返回出错号。
        dir = dir_namei(newname,&namelen,&basename);
        if (!dir) {
                iput(oldinode);
                return -EACCES;
        }
// 如果新路径名中不包括文件名，则释放原路径名 i 节点和新路径名目录的 i 节点，返回出错号。
        if (!namelen) {
                iput(oldinode);
                iput(dir);
                return -EPERM;
        }
// 如果新路径名目录的设备号与原路径名的设备号不一样，则也不能建立连接，于是释放新路径名
// 目录的 i 节点和原路径名的 i 节点，返回出错号。
        if (dir->i_dev != oldinode->i_dev) {
                iput(dir);
                iput(oldinode);
                return -EXDEV;
        }
// 如果用户没有在新目录中写的权限，则也不能建立连接，于是释放新路径名目录的 i 节点和原路径名
// 的 i 节点，返回出错号。
```

```
        if (!permission(dir,MAY_WRITE)) {
                iput(dir);
                iput(oldinode);
                return -EACCES;
        }
// 查询该新路径名是否已经存在，如果存在，则也不能建立连接，于是释放包含该已存在目录项的高速
// 缓冲区，释放新路径名目录的 i 节点和原路径名的 i 节点，返回出错号。
        bh = find_entry(&dir,basename,namelen,&de);
        if (bh) {
                brelse(bh);
                iput(dir);
                iput(oldinode);
                return -EEXIST;
        }
// 在新目录中添加一个目录项。若失败则释放该目录的 i 节点和原路径名的 i 节点，返回出错号。
        bh = add_entry(dir,basename,namelen,&de);
        if (!bh) {
                iput(dir);
                iput(oldinode);
                return -ENOSPC;
        }
// 否则初始设置该目录项的 i 节点号等于原路径名的 i 节点号，并置包含该新添目录项的高速缓冲区
// 已修改标志，释放该缓冲区，释放目录的 i 节点。
        de->inode = oldinode->i_num;
        bh->b_dirt = 1;
        brelse(bh);
        iput(dir);
// 将原节点的应用计数加 1，修改其改变时间为当前时间，并设置 i 节点已修改标志，最后释放原
// 路径名的 i 节点，并返回 0(成功)。
        oldinode->i_nlinks++;
        oldinode->i_ctime = CURRENT_TIME;
        oldinode->i_dirt = 1;
        iput(oldinode);
        return 0;
}
```

9.18 char_dev.c 文件

9.18.1 功能描述

9.18.2 代码注释

列表 linux/fs/char_dev.c 程序

```
/*
 *  linux/fs/char_dev.c
 *
 *  (C) 1991  Linus Torvalds
```

```
 */

#include <errno.h>         // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
#include <sys/types.h>     // 类型头文件。定义了基本的系统数据类型。

#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/kernel.h>  // 内核头文件。含有一些内核常用函数的原形定义。

#include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
#include <asm/io.h>        // io 头文件。定义硬件端口输入/输出宏汇编语句。

extern int tty_read(unsigned minor,char * buf,int count);      // 终端读。
extern int tty_write(unsigned minor,char * buf,int count);     // 终端写。

// 定义字符设备读写函数指针类型。
typedef (*crw_ptr)(int rw,unsigned minor,char * buf,int count,off_t * pos);

//// 终端读写操作函数。
// 参数：rw - 读写命令；minor - 终端子设备号；buf - 缓冲区；cout - 读写字节数；
//       pos - 读写操作当前指针，对于终端操作，该指针无用。
// 返回：实际读写的字节数。
static int rw_ttyx(int rw,unsigned minor,char * buf,int count,off_t * pos)
{
        return ((rw==READ)?tty_read(minor,buf,count):
                tty_write(minor,buf,count));
}

//// 终端读写操作函数。
// 同上 rw_ttyx()，只是增加了对进程是否有控制终端的检测。
static int rw_tty(int rw,unsigned minor,char * buf,int count, off_t * pos)
{
// 若进程没有对应的控制终端，则返回出错号。
        if (current->tty<0)
                return -EPERM;
// 否则调用终端读写函数 rw_ttyx()，并返回实际读写字节数。
        return rw_ttyx(rw,current->tty,buf,count,pos);
}

//// 内存数据读写。未实现。
static int rw_ram(int rw,char * buf, int count, off_t *pos)
{
        return -EIO;
}

//// 内存数据读写操作函数。未实现。
static int rw_mem(int rw,char * buf, int count, off_t * pos)
{
        return -EIO;
}

//// 内核数据区读写函数。未实现。
static int rw_kmem(int rw,char * buf, int count, off_t * pos)
```

```
{
        return -EIO;
}

// 端口读写操作函数。
// 参数：rw - 读写命令；buf - 缓冲区；cout - 读写字节数；pos - 端口地址。
// 返回：实际读写的字节数。
static int rw_port(int rw,char * buf, int count, off_t * pos)
{
        int i=*pos;

// 对于所要求读写的字节数，并且端口地址小于 64k 时，循环执行单个字节的读写操作。
        while (count-->0 && i<65536) {
// 若是读命令，则从端口 i 中读取一字节内容并放到用户缓冲区中。
                if (rw==READ)
                        put_fs_byte(inb(i),buf++);
// 若是写命令，则从用户数据缓冲区中取一字节输出到端口 i。
                else
                        outb(get_fs_byte(buf++),i);
// 前移一个端口。[??]
                i++;
        }
// 计算读/写的字节数，并相应调整读写指针。
        i -= *pos;
        *pos += i;
// 返回读/写的字节数。
        return i;
}

//// 内存读写操作函数。
static int rw_memory(int rw, unsigned minor, char * buf, int count, off_t * pos)
{
// 根据内存设备子设备号，分别调用不同的内存读写函数。
        switch(minor) {
                case 0:
                        return rw_ram(rw,buf,count,pos);
                case 1:
                        return rw_mem(rw,buf,count,pos);
                case 2:
                        return rw_kmem(rw,buf,count,pos);
                case 3:
                        return (rw==READ)?0:count;      /* rw_null */
                case 4:
                        return rw_port(rw,buf,count,pos);
                default:
                        return -EIO;
        }
}

// 定义系统中设备种数。
#define NRDEVS ((sizeof (crw_table))/(sizeof (crw_ptr)))

// 字符设备读写函数指针表。
```

```
static crw_ptr crw_table[]={
        NULL,           /* nodev */             /* 无设备(空设备) */
        rw_memory,      /* /dev/mem etc */      /* /dev/mem 等 */
        NULL,           /* /dev/fd */           /* /dev/fd 软驱 */
        NULL,           /* /dev/hd */           /* /dev/hd 硬盘 */
        rw_ttyx,        /* /dev/ttyx */         /* /dev/ttyx 串口终端 */
        rw_tty,         /* /dev/tty */          /* /dev/tty 终端 */
        NULL,           /* /dev/lp */           /* /dev/lp 打印机 */
        NULL};          /* unnamed pipes */     /* 未命名管道 */

//// 字符设备读写操作函数。
// 参数：rw - 读写命令；dev - 设备号；buf - 缓冲区；count - 读写字节数；pos -读写指针。
// 返回：实际读/写字节数。
int rw_char(int rw,int dev, char * buf, int count, off_t * pos)
{
        crw_ptr call_addr;

// 如果设备号超出系统设备数，则返回出错码。
        if (MAJOR(dev)>=NRDEVS)
                return -ENODEV;
// 若该设备没有对应的读/写函数，则返回出错码。
        if (!(call_addr=crw_table[MAJOR(dev)]))
                return -ENODEV;
// 调用对应设备的读写操作函数，并返回实际读/写的字节数。
        return call_addr(rw,MINOR(dev),buf,count,pos);
}

```

9.19 ioctl.c 文件

9.19.1 功能描述

9.19.2 代码注释

列表 linux/fs/ioctl.c 程序

```
/*
 *  linux/fs/ioctl.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
#include <errno.h>         // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
#include <sys/stat.h>      // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。

#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。

```

```
extern int tty_ioctl(int dev, int cmd, int arg);  // 终端 ioctl(chr_drv/tty_ioctl.c, 115)。

// 定义输入输出控制(ioctl)函数指针。
typedef int (*ioctl_ptr)(int dev,int cmd,int arg);

// 定义系统中设备种数。
#define NRDEVS ((sizeof (ioctl_table))/(sizeof (ioctl_ptr)))

// ioctl 操作函数指针表。
static ioctl_ptr ioctl_table[]={
        NULL,           /* nodev */
        NULL,           /* /dev/mem */
        NULL,           /* /dev/fd */
        NULL,           /* /dev/hd */
        tty_ioctl,      /* /dev/ttyx */
        tty_ioctl,      /* /dev/tty */
        NULL,           /* /dev/lp */
        NULL};          /* named pipes */


//// 系统调用函数 - 输入输出控制函数。
// 参数：fd - 文件描述符；cmd - 命令码；arg - 参数。
// 返回：成功则返回 0，否则返回出错码。
int sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
        struct file * filp;
        int dev,mode;

// 如果文件描述符超出可打开的文件数，或者对应描述符的文件结构指针为空，则返回出错码，退出。
        if (fd >= NR_OPEN || !(filp = current->filp[fd]))
                return -EBADF;
// 取对应文件的属性。如果该文件不是字符文件，也不是块设备文件，则返回出错码，退出。
        mode=filp->f_inode->i_mode;
        if (!S_ISCHR(mode) && !S_ISBLK(mode))
                return -EINVAL;
// 从字符或块设备文件的 i 节点中取设备号。如果设备号大于系统现有的设备数，则返回出错号。
        dev = filp->f_inode->i_zone[0];
        if (MAJOR(dev) >= NRDEVS)
                return -ENODEV;
// 如果该设备在 ioctl 函数指针表中没有对应函数，则返回出错码。
        if (!ioctl_table[MAJOR(dev)])
                return -ENOTTY;
// 否则返回实际 ioctl 函数返回码，成功则返回 0，否则返回出错码。
        return ioctl_table[MAJOR(dev)](dev,cmd,arg);
}
```

第 10 章 内存管理(mm)

10.1 概述

内存管理目录中共有三个程序，如下表所示：

列表 10.1 内存管理子目录文件列表

Name	Size	Last modified (GMT)	Description
Makefile	813 bytes	1991-12-02 03:21:45	m
memory.c	11223 bytes	1991-12-03 00:48:01	m
page.s	508 bytes	1991-10-02 14:16:30	m

下面对每一个文件分别描述。

10.2 Makefile 文件

10.2.1 功能描述

10.2.2 代码注释

列表 10.2 linux/mm/Makefile 文件

```
CC      =gcc     # GNU C 语言编译器。
CFLAGS  =-O -Wall -fstrength-reduce -fcombine-regs -fomit-frame-pointer \
        -finline-functions -nostdinc -I../include
  # C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
  # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
  # 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
  # 单短小的函数代码嵌入调用程序中；-nostdinc -I../include 不使用默认路径中的包含文件，而
  # 使用这里指定目录中的(../include)。
AS      =gas     # GNU 的汇编程序。
AR      =gar     # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。
LD      =gld     # GNU 的连接程序。
CPP     =gcc -E -nostdinc -I../include
  # C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
  # 出设备或指定的输出文件中；-nostdinc -I../include 同前。

  # 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
  # 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
  # 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
```

```
    # 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。其中$*.s（或$@）是自动目标变量，
    # $<代表第一个先决条件，这里即是符合条件*.c 的文件。
9 .c.o:
10         $(CC) $(CFLAGS) \
11         -c -o $*.o $<
  # 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。22 行是实现该操作的具体命令。
12 .s.o:
13         $(AS) -o $*.o $<
14 .c.s:                        # 类似上面，*.c 文件-→*.s 汇编程序文件。不进行连接。
15         $(CC) $(CFLAGS) \
16         -S -o $*.s $<
17
18 OBJS    = memory.o page.o    # 定义目标文件变量 OBJS。
19
20 all: mm.o
21
22 mm.o: $(OBJS)          # 在有了先决条件 OBJS 后使用下面的命令连接成目标 mm.o
23         $(LD) -r -o mm.o $(OBJS)
24
    # 下面的规则用于清理工作。当执行'make clean'时，就会执行 26--27 行上的命令，去除所有编译
    # 连接生成的文件。'rm'是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
25 clean:
26         rm -f core *.o *.a tmp_make
27         for i in *.c;do rm -f `basename $$i .c`.s;done
28
    # 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
    # 使用字符串编辑程序 sed 对 Makefile 文件（这里即是自己）进行处理，输出为删除 Makefile
    # 文件中'### Dependencies'行后面的所有行（下面从 35 开始的行），并生成 tmp_make
    # 临时文件（30 行的作用）。然后对 mm/目录下的每一个 C 文件执行 gcc 预处理操作.
    # -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
    # 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
    # 文件名加上其依赖关系--该源文件中包含的所有头文件列表。把预处理结果都添加到临时
    # 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。

29 dep:
30         sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
31         (for i in *.c;do $(CPP) -M $$i;done) >> tmp_make
32         cp tmp_make Makefile
33
34 ### Dependencies:
35 memory.o : memory.c ../include/signal.h ../include/sys/types.h \
36   ../include/asm/system.h ../include/linux/sched.h ../include/linux/head.h \
37   ../include/linux/fs.h ../include/linux/mm.h ../include/linux/kernel.h
```

10.3 memory.c 程序

10.3.1 功能描述

10.3.2 代码注释

列表 linux/mm/memory.c 程序

```
/*
 *  linux/mm/memory.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * demand-loading started 01.12.91 - seems it is high on the list of
 * things wanted, and it should be easy to implement. - Linus
 */
/*
 * 需求加载是从 01.12.91 开始编写的 - 在程序编制表中是呼是最重要的程序，
 * 并且应该是很容易编制的 - linus
 */

/*
 * Ok, demand-loading was easy, shared pages a little bit tricker. Shared
 * pages started 02.12.91, seems to work. - Linus.
 *
 * Tested sharing by executing about 30 /bin/sh: under the old kernel it
 * would have taken more than the 6M I have free, but it worked well as
 * far as I could see.
 *
 * Also corrected some "invalidate()"s - I wasn't doing enough of them.
 */
/*
 * OK，需求加载是比较容易编写的，而共享页面却需要点技巧。共享页面程序是
 * 02.12.91 开始编写的，好象能够工作 - Linus。
 *
 * 通过执行大约 30 个/bin/sh 对共享操作进行了测试：在老内核当中需要占用多于
 * 6M 的内存，而目前却不用。目前看来工作的很好。
 *
 * 对"invalidate()"函数也进行了修正 - 在这方面我还做的不够。
 */

#include <signal.h>        // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。

#include <asm/system.h>    // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

#include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                           // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
#include <linux/head.h>    // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
#include <linux/kernel.h>  // 内核头文件。含有一些内核常用函数的原形定义。

volatile void do_exit(long code);

//// 内存已用完。
static inline volatile void oom(void)
{
        printk("out of memory\n\r");
```

```
        do_exit(SIGSEGV);
}

// 刷新页变换高速缓冲。
// 为了提高地址转换的效率，CPU 将最近使用的页表数据存放在芯片中高速缓冲中。在修改过页表
// 信息之后，就需要刷新该缓冲区。这里使用重新加载页目录基址寄存器 cr3 的方法来进行刷新。
#define invalidate() \
__asm__("movl %%eax,%%cr3"::"a" (0))

/* these are not to be changed without changing head.s etc */
/* 下面代码需要与 head.s 等中的相关信息一起改变 */
#define LOW_MEM 0x100000                    // 内存低端（1MB）。
#define PAGING_MEMORY (15*1024*1024)        // 分页内存 15MB。
#define PAGING_PAGES (PAGING_MEMORY>>12)    // 分页后的页数。
#define MAP_NR(addr) (((addr)-LOW_MEM)>>12) // 映射页号。
#define USED 100

// CODE_SPACE(addr)  ((((addr)+0xfff)&~0xfff) < current->start_code + current->end_code。
#define CODE_SPACE(addr) ((((addr)+4095)&~4095) < \
current->start_code + current->end_code)

static long HIGH_MEMORY = 0;

// 复制页面（4K 字节）。
#define copy_page(from,to) \
__asm__("cld ; rep ; movsl"::"S" (from),"D" (to),"c" (1024):"cx","di","si")

static unsigned char mem_map [ PAGING_PAGES ] = {0,};  // 内存映射字节图(1 字节代表 1 页)。

/*
 * Get physical address of first (actually last :-) free page, and mark it
 * used. If no free pages left, return 0.
 */
/*
 * 获取首个(实际上是最后 1 个:-)空闲页面，并标记为已使用。如果没有空闲页面了，
 * 就返回 0。
 */
//// 取空闲页面。如果已经没有内存了，则返回 0。
// 输入：%1(ax=0) - 0；%2(LOW_MEM)；%3(cx=PAGING PAGES)；%4(di=mem_map+PAGING_PAGES-1)。
// 输出：返回%0(ax=页面号)。
// 从内存映像末端开始向前扫描所有页面标志（页面总数为 PAGING_PAGES），如果有页面空闲（对应
// 内存映像位为 0）则返回页面地址。
unsigned long get_free_page(void)
{
register unsigned long __res asm("ax");

__asm__("std ; repne ; scasb\n\t"       // 方向位置位，将 al(0)与对应每个页面的(di)内容比较，
        "jne 1f\n\t"                    // 如果没有等于 0 的字节，则跳转结束（返回 0）。
        "movb $1,1(%%edi)\n\t           // 将对应页面的内存映像位置 1。
        "sall $12,%%ecx\n\t"            // 页面数*4K = 相对页面起始地址。
        "addl %2,%%ecx\n\t"             // 再加上低端内存地址，即获得页面实际起始地址。
        "movl %%ecx,%%edx\n\t"          // 将页面实际起始地址➔edx 寄存器。
        "movl $1024,%%ecx\n\t"          // 寄存器 ecx 置计数值 1024。
```

```
        "leal 4092(%%edx),%%edi\n\t"  // 将 4092+edx 的位置➔edi(该页面的末端)。
        "rep ; stosl\n\t"             // 将 edi 所指内存清零（反方向，也即将该页面清零）。
        "movl %%edx,%%eax\n"          // 将页面起始地址➔eax（返回值）。
        "1:"
        :"=a" (__res)
        :"" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
        "D" (mem_map+PAGING_PAGES-1)
        :"di","cx","dx");
return __res;                         // 返回空闲页面地址（如果无空闲也则返回 0）。
}

/*
 * Free a page of memory at physical address 'addr'. Used by
 * 'free_page_tables()'
 */
/*
 * 释放物理地址'addr'的页面。用于'free_page_tables()'。
 */
//// 释放物理地址 addr 开始的一页面内存。
// 1MB 以下的内存空间用于操作系统（内核），不作为分配页面的内存。
void free_page(unsigned long addr)
{
        if (addr < LOW_MEM) return;   // 如果物理地址 addr 小于内存低端（1MB），则返回。
        if (addr >= HIGH_MEMORY)      // 如果物理地址 addr>=内存高端，则显示出错信息。
                panic("trying to free nonexistent page");
        addr -= LOW_MEM;              // 物理地址-低端内存位置，再除以 4KB，得页面号。
        addr >>= 12;
        if (mem_map[addr]--) return;  // 如果对应内存页面映射字节不等于 0，则减 1 返回。
        mem_map[addr]=0;              // 否则置对应页面映射字节为 0，并显示出错信息，死机。
        panic("trying to free free page");
}

/*
 * This function frees a continuos block of page tables, as needed
 * by 'exit()'. As does copy_page_tables(), this handles only 4Mb blocks.
 */
/*
 * 下面函数释放页表连续的内存块，'exit()'需要该函数。与 copy_page_tables()
 * 类似，该函数仅处理 4Mb 的内存块。
 */
//// 根据指定的地址和限长，释放对应内存页表所指定的内存块及页表本身。
// 页目录位于物理地址 0 开始处，共 1024 项，占 4K 字节。每个目录项指定一个页表。
// 页表从物理地址 0x1000 处开始（紧接着目录空间），每个页表有 1024 项，也占 4K 内存。
// 每个页表项指定一页内存（4K）。目录项和页表项的大小均为 4 个字节。
// 参数：from - 起始基地址；size - 释放的长度。
int free_page_tables(unsigned long from,unsigned long size)
{
        unsigned long *pg_table;
        unsigned long * dir, nr;

        if (from & 0x3fffff)                  // 要释放内存块的地址需以 4M 为边界。
                panic("free_page_tables called with wrong alignment");
        if (!from)                            // 出错，试图释放内核所占空间。
```

```
                panic("Trying to free up swapper memory space");
// 计算所占页目录项数(4M 的进位整数倍)，也即所占页表数。
        size = (size + 0x3fffff) >> 22;
// 下面一句计算起始目录项。对应的目录项号=from>>22，因每项占 4 字节，又由于页目录是从
// 物理地址 0 开始，因此实际的目录项指针=目录项号<<2，也即(from>>20)。与上 0xffc 确保
// 目录项指针范围有效。
        dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
        for ( ; size-->0 ; dir++) {              // size 现在是需释放内存的目录项数。
                if (!(1 & *dir))                 // 如果该目录项无效(P 位=0)，则继续。
                        continue;
                pg_table = (unsigned long *) (0xfffff000 & *dir);  // 取目录项中页表地址。
                for (nr=0 ; nr<1024 ; nr++) {    // 每个页表有 1024 个页项。
                        if (1 & *pg_table)       // 如果该页表项有效(P 位=1)，则释放该页。
                                free_page(0xfffff000 & *pg_table);
                        *pg_table = 0;           // 该页表项内容清零。
                        pg_table++;              // 指向页表中下一项。
                }
                free_page(0xfffff000 & *dir);    // 释放该页表所占内存页面。
                *dir = 0;                        // 对相应页表的目录项清零。
        }
        invalidate();                            // 刷新页变换高速缓冲。
        return 0;
}

/*
 *  Well, here is one of the most complicated functions in mm. It
 * copies a range of linerar addresses by copying only the pages.
 * Let's hope this is bug-free, 'cause this one I don't want to debug :-)
 *
 * Note! We don't copy just any chunks of memory - addresses have to
 * be divisible by 4Mb (one page-directory entry), as this makes the
 * function easier. It's used only by fork anyway.
 *
 * NOTE 2!! When from==0 we are copying kernel space for the first
 * fork(). Then we DONT want to copy a full page-directory entry, as
 * that would lead to some serious memory waste - we just copy the
 * first 160 pages - 640kB. Even that is more than we need, but it
 * doesn't take any more memory - we don't copy-on-write in the low
 * 1 Mb-range, so the pages can be shared with the kernel. Thus the
 * special case for nr=xxxx.
 */
/*
 * 好了，下面是内存管理 mm 中最为复杂的程序之一。它通过只复制内存页面
 * 来拷贝一定范围内的线性地址中的内容。希望代码中没有错误，因为我不想
 * 再调试这块代码了☺。
 *
 * 注意！我们并不是仅复制任何内存块 - 内存块的地址需要是 4Mb 的倍数（正好
 * 一个页目录项对应的内存大小），因为这样处理可使函数很简单。不管怎样，
 * 它仅被 fork()使用。
 *
 * 注意 2！！当 from==0 时，是在为第一次 fork()调用复制内核空间。此时我们
 * 不想复制整个页目录项对应的内存，因为这样做会导致严重的内存浪费 - 我们
 * 只复制头 160 个页面 - 对应 640kB。即使是复制这些页面也已经超出我们的需求，
```

```
 * 但这不会占用更多的内存 - 在低 1Mb 内存范围内我们不执行写时复制操作，所以
 * 这些页面可以与内核共享。因此这是 nr=xxxx 的特殊情况。
 */
//// 复制进程的页目录页表。
int copy_page_tables(unsigned long from,unsigned long to,long size)
{
        unsigned long * from_page_table;
        unsigned long * to_page_table;
        unsigned long this_page;
        unsigned long * from_dir, * to_dir;
        unsigned long nr;

// 源地址和目的地址都需要是 4Mb 的倍数。否则出错，死机。
        if ((from&0x3fffff) || (to&0x3fffff))
                panic("copy_page_tables called with wrong alignment");
// 取得源地址和目的地址的目录项(from_dir 和 to_dir)。
        from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
        to_dir = (unsigned long *) ((to>>20) & 0xffc);
// 计算要复制的内存块占用的页表数（也即目录项数）。
        size = ((unsigned) (size+0x3fffff)) >> 22;
// 下面开始对每个占用的页表依次进行复制操作。
        for( ; size-->0 ; from_dir++,to_dir++) {
// 如果目的目录项指定的页表已经存在(P=1)，则出错，死机。
                if (1 & *to_dir)
                        panic("copy_page_tables: already exist");
// 如果此源目录项未被使用，则不用复制对应页表，跳过。
                if (!(1 & *from_dir))
                        continue;
// 取当前源目录项中页表的地址➔from_page_table。
                from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
// 为目的页表取一页空闲内存，如果返回是 0 则说明没有申请到空闲内存页面。返回值=-1，退出。
                if (!(to_page_table = (unsigned long *) get_free_page()))
                        return -1;      /* Out of memory, see freeing */
// 设置目的目录项信息。7 是标志信息，表示(Usr, R/W, Present)。
                *to_dir = ((unsigned long) to_page_table) | 7;
// 针对当前处理的页表，设置需复制的页面数。如果是在内核空间，则仅需复制头 160 页，否则需要
// 复制 1 个页表中的所有 1024 页面。
                nr = (from==0)?0xA0:1024;
// 对于当前页表，开始复制指定数目 nr 个内存页面。
                for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
                        this_page = *from_page_table;  // 取源页表项内容。
                        if (!(1 & this_page))       // 如果当前源页面没有使用，则不用复制。
                                continue;
// 复位页表项中 R/W 标志(置 0)。(如果 U/S 位是 0，则 R/W 就没有作用。如果 U/S 是 1，而 R/W 是 0，
// 那么运行在用户层的代码就只能读页面。如果 U/S 和 R/W 都置位，则就有写的权限。)
                        this_page &= ~2;
                        *to_page_table = this_page; // 将该页表项复制到目的页表中。
// 如果该页表项所指页面的地址在 1M 以上，则需要设置内存页面映射数组 mem_map[]，于是计算
// 页面号，并以它为索引在页面映射数组相应项中增加引用次数。
                        if (this_page > LOW_MEM) {
                                *from_page_table = this_page; // 令源页表项也只读[??]。
                                this_page -= LOW_MEM;
                                this_page >>= 12;
```

```
                                        mem_map[this_page]++;
                                }
                        }
                }
                invalidate();                                   // 刷新页变换高速缓冲。
                return 0;
}

/*
 * This function puts a page in memory at the wanted address.
 * It returns the physical address of the page gotten, 0 if
 * out of memory (either when trying to access page-table or
 * page.)
 */
/*
 * 下面函数将一页面放置在内存中指定地址处。它返回页面的物理地址，如果
 * 内存不够(在访问页表或页面时)，则返回 0。
 */
//// 在指定物理地址处放置一页面。
// 主要工作是在页目录和页表中设置指定页面的信息。
// 若成功则返回页面地址。
unsigned long put_page(unsigned long page,unsigned long address)
{
        unsigned long tmp, *page_table;

/* NOTE !!! This uses the fact that _pg_dir=0 */
/* 注意!!!这里使用了页目录基址_pg_dir=0 的条件 */

// 如果申请的页面位置低于 LOW_MEM(1Mb)或超出系统实际含有内存高端 HIGH_MEMORY，则发出警告。
        if (page < LOW_MEM || page >= HIGH_MEMORY)
                printk("Trying to put page %p at %p\n",page,address);
// 如果申请的页面在内存页面映射字节图中没有置位，则显示警告信息。
        if (mem_map[(page-LOW_MEM)>>12] != 1)
                printk("mem_map disagrees with %p at %p\n",page,address);
// 计算指定地址在页目录中对应的目录项指针。
        page_table = (unsigned long *) ((address>>20) & 0xffc);
// 如果该目录项有效(P=1)(也即指定的页表在内存中)，则从中取得指定页表的地址➔page_table。
        if ((*page_table)&1)
                page_table = (unsigned long *) (0xfffff000 & *page_table);
        else {
// 否则，申请空闲页面给页表使用，并在对应目录项中置相应标志 7（User, U/S, R/W）。然后将
// 该页表的地址➔page_table。
                if (!(tmp=get_free_page()))
                        return 0;
                *page_table = tmp|7;
                page_table = (unsigned long *) tmp;
        }
// 在页表中设置指定地址页面的页表项内容。每个页表共可有 1024 项(0x3ff)。
        page_table[(address>>12) & 0x3ff] = page | 7;
/* no need for invalidate */
/* 不需要刷新页变换高速缓冲 */
        return page;            // 返回页面地址。
}
```

```
//// 取消写保护页面函数（写时复制）。
// 输入参数为页表项指针。
// [?? un_wp_page 意思是否是取消页面的写保护?? wp -- Write Protected。]
void un_wp_page(unsigned long * table_entry)
{
        unsigned long old_page,new_page;

        old_page = 0xfffff000 & *table_entry;  // 取原页面地址。
// 如果原页面地址大于内存低端 LOW_MEM(1Mb)并且其在页面映射字节数组中值为 1（表示仅被
// 引用 1 次，页面没有被共享），则在该页面的页表项中置 R/W 标志（可写），并刷新页变换
// 高速缓冲，然后返回。
        if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
                *table_entry |= 2;
                invalidate();
                return;
        }
// 否则，申请一页空闲页面。
        if (!(new_page=get_free_page()))
                oom();                          // Out of Memory。内存不够处理。
// 如果原页面大于内存低端（则意味着 mem_map[]>1，页面是共享的），则将原页面的页面映射
// 数组值递减 1。然后将指定页表项内容更新为新页面的地址，并置可读写等标志(U/S, R/W, P)。
// 刷新页变换高速缓冲。最后将原页面内容复制到新页面。
        if (old_page >= LOW_MEM)
                mem_map[MAP_NR(old_page)]--;
        *table_entry = new_page | 7;
        invalidate();
        copy_page(old_page,new_page);
}

/*
 * This routine handles present pages, when users try to write
 * to a shared page. It is done by copying the page to a new address
 * and decrementing the shared-page counter for the old page.
 *
 * If it's in code space we exit with a segment error.
 */
/*
 * 当用户试图往一个共享页面上写时，该函数处理已存在的内存页面，（写时复制）
 * 它是通过将页面复制到一个新地址上并递减原页面的共享页面计数值实现的。
 *
 * 如果它在代码空间，我们就以段错误信息退出。
 */
//// 页面异常中断处理调用函数。写共享页面处理函数。
// 参数 error_code 是由 CPU 自动产生，address 是页面线性地址。
// 写共享页面时，需复制页面（写时复制）。
void do_wp_page(unsigned long error_code,unsigned long address)
{
#if 0
/* we cannot do this yet: the estdio library writes to code space */
/* stupid, stupid. I really want the libc.a from GNU */
/* 我们现在还不能这样做：因为 estdio 库会在代码空间执行写操作 */
/* 真是太愚蠢了。我真想从 GNU 得到 libc.a 库。*/
```

```
        if (CODE_SPACE(address))      // 如果地址位于代码空间，则终止执行程序。
                do_exit(SIGSEGV);
#endif
// 处理取消页面保护。参数指定页面在页表中的页表项指针，其计算方法是：
// ((address>>10) & 0xffc)：计算指定地址的页面在页表中的偏移地址；
// (0xfffff000 &((address>>20) &0xffc))：取目录项中页表的地址值，
// 其中((address>>20) &0xffc)计算页面所在页表的目录项指针；
// 两者相加即得指定地址对应页面的页表项指针。
        un_wp_page((unsigned long *)
                (((address>>10) & 0xffc) + (0xfffff000 &
                *((unsigned long *) ((address>>20) &0xffc)))));

}

//// 写页面验证。
void write_verify(unsigned long address)
{
        unsigned long page;

// 判断指定地址所对应的页表页面是否存在(P)，若不存在(P=0)则返回。
        if (!( (page = *((unsigned long *) ((address>>20) & 0xffc)) )&1))
                return;
// 取页表的地址，加上指定地址的页面在页表中的页表项偏移值，得对应页面的页表项指针。
        page &= 0xfffff000;
        page += ((address>>10) & 0xffc);
// 如果该页面不可写(标志 R/W 没有置位)，则执行共享检验和复制页面操作（写时复制）。
        if ((3 & *(unsigned long *) page) == 1)  /* non-writeable, present */
                un_wp_page((unsigned long *) page);
        return;
}

//// 取指定地址的空页面。
// 与 get_free_page()不同。get_free_page()仅置内存页面映射数组 mem_map[]中的引用标志。
// 而这里 get_empty_page()不仅是取指定地址处的页面，还进一步调用 put_page()，将页面信息
// 添加到页目录和页表中。
void get_empty_page(unsigned long address)
{
        unsigned long tmp;

        if (!(tmp=get_free_page()) || !put_page(tmp,address)) {
                free_page(tmp);         /* 0 is ok - ignored */
                oom();
        }
}

/*
 * try_to_share() checks the page at address "address" in the task "p",
 * to see if it exists, and if it is clean. If so, share it with the current
 * task.
 *
 * NOTE! This assumes we have checked that p != current, and that they
 * share the same executable.
 */
```

```
/*
 * try_to_share()在任务"p"中检查位于地址"address"处的页面，看页面是否存在，是否干净。
 * 如果是干净的话，就与当前任务共享。
 *
 * 注意！这里我们已假定 p !=当前任务，并且它们共享同一个执行程序。
 */
//// 尝试共享。
// 返回 1-成功，0-失败。
static int try_to_share(unsigned long address, struct task_struct * p)
{
        unsigned long from;
        unsigned long to;
        unsigned long from_page;
        unsigned long to_page;
        unsigned long phys_addr;

// 求指定内存地址的页目录项。
        from_page = to_page = ((address>>20) & 0xffc);
// 计算地址在指定进程 p 中对应的页目录项。
        from_page += ((p->start_code>>20) & 0xffc);
// 计算地址在当前进程中对应的页目录项。
        to_page += ((current->start_code>>20) & 0xffc);
/* is there a page-directory at from? */
/* 在 from 处是否存在页目录？*/
// *** 对 p 进程页面进行操作。
// 取页目录项内容。如果该目录项无效(P=0)，则返回。否则取该目录项对应页表地址➔from。
        from = *(unsigned long *) from_page;
        if (!(from & 1))
                return 0;
        from &= 0xfffff000;
// 计算地址对应的页表项指针值，并取出该页表项内容➔phys_addr。
        from_page = from + ((address>>10) & 0xffc);
        phys_addr = *(unsigned long *) from_page;
/* is the page clean and present? */
/* 页面干净并且存在吗？*/
// 0x41 对应页表项中的 Dirty 和 Present 标志。如果页面不干净或无效则返回。
        if ((phys_addr & 0x41) != 0x01)
                return 0;
// 取页面的地址➔phys_addr。如果该页面地址不存在或小于内存低端(1M)也返回退出。
        phys_addr &= 0xfffff000;
        if (phys_addr >= HIGH_MEMORY || phys_addr < LOW_MEM)
                return 0;
// *** 对当前进程页面进行操作。
// 取页目录项内容➔to。如果该目录项无效(P=0)，则取空闲页面，并让更新 to_page 所指的目录项。
        to = *(unsigned long *) to_page;
        if (!(to & 1))
                if (to = get_free_page())
                        *(unsigned long *) to_page = to | 7;
                else
                        oom();
// 取对应页表地址➔to，页表项地址➔to_page。如果对应的页面已经存在，则出错，死机。
        to &= 0xfffff000;
        to_page = to + ((address>>10) & 0xffc);
```

```
        if (1 & *(unsigned long *) to_page)
                panic("try_to_share: to_page already exists");
/* share them: write-protect */
/* 对它们进行共享处理：写保护 */
// 对 p 进程中页面置写保护标志(置 R/W=0)。并且当前进程中的对应页表项指向它。
        *(unsigned long *) from_page &= ~2;
        *(unsigned long *) to_page = *(unsigned long *) from_page;
// 刷新页变换高速缓冲。
        invalidate();
// 计算所操作页面的页面号，并将对应页面映射数组项中的引用递增 1。
        phys_addr -= LOW_MEM;
        phys_addr >>= 12;
        mem_map[phys_addr]++;
        return 1;
}

/*
 * share_page() tries to find a process that could share a page with
 * the current one. Address is the address of the wanted page relative
 * to the current data space.
 *
 * We first check if it is at all feasible by checking executable->i_count.
 * It should be >1 if there are other tasks sharing this inode.
 */
/*
 * share_page()试图找到一个进程，它可以与当前进程共享页面。参数 address 是
 * 当前数据空间中期望共享的某页面地址。
 *
 * 首先我们通过检测 executable->i_count 来查证是否可行。如果有其它任务已共享
 * 该 inode，则它应该大于 1。
 */
//// 共享页面。
// 返回 1 - 成功，0 - 失败。
static int share_page(unsigned long address)
{
        struct task_struct ** p;

// 如果是不可执行的，则返回。
        if (!current->executable)
                return 0;
// 如果只能单独执行(executable->i_count=1)，也退出。
        if (current->executable->i_count < 2)
                return 0;
// 搜索任务数组中所有任务。寻找与当前进程可共享页面的进程，并尝试对指定地址的页面进行共享。
        for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
                if (!*p)                        // 如果该任务项空闲，则继续寻找。
                        continue;
                if (current == *p)              // 如果就是当前任务，也继续寻找。
                        continue;
                if ((*p)->executable != current->executable) // 如果 executable 不等，也继续。
                        continue;
                if (try_to_share(address,*p)) // 尝试共享页面。
                        return 1;
```

```
        }
        return 0;
}

//// 页面异常中断处理调用函数。缺页异常处理函数。
// 参数 error_code 是由 CPU 自动产生，address 是页面线性地址。
void do_no_page(unsigned long error_code,unsigned long address)
{
        int nr[4];
        unsigned long tmp;
        unsigned long page;
        int block,i;

        address &= 0xfffff000;          // 页面地址。
// 如果当前进程的 executable 空，或者指定地址超出进程的数据末端，则取指定地址空页面并退出。
        tmp = address - current->start_code;
        if (!current->executable || tmp >= current->end_data) {
                get_empty_page(address);
                return;
        }
// 如果在所有进程中尝试共享页面失败，则退出。
        if (share_page(tmp))
                return;
// 取空闲页面，如果内存不够了，则进行内存不够处理。
        if (!(page = get_free_page()))
                oom();
/* remember that 1 block is used for header */
/* 记住，（任务）头要使用 1 个数据块 */
//
        block = 1 + tmp/BLOCK_SIZE;
        for (i=0 ; i<4 ; block++,i++)
                nr[i] = bmap(current->executable,block);
        bread_page(page,current->executable->i_dev,nr);
        i = tmp + 4096 - current->end_data;
        tmp = page + 4096;
        while (i-- > 0) {
                tmp--;
                *(char *)tmp = 0;
        }
        if (put_page(page,address))
                return;
        free_page(page);
        oom();
}

//// 内存初始化。
// 参数：start_mem - 可用作分页处理的物理内存起始位置（已去除 RAMDISK 所占内存空间等）。
//        end_mem   - 实际物理内存最大地址。
// 在该版的 linux 内核中，最多能使用 16Mb 的内存，大于 16Mb 的内存将不于考虑，弃置不用。
// 0 - 1Mb 内存空间用于内核系统（其实是 0-640Kb）。
void mem_init(long start_mem, long end_mem)
{
        int i;
```

```
        HIGH_MEMORY = end_mem;             // 设置内存最高端。
        for (i=0 ; i<PAGING_PAGES ; i++)   // 首先置所有页面为已占用(USED=100)状态,
                mem_map[i] = USED;         // 即将页面映射数组全置成 USED。
        i = MAP_NR(start_mem);             // 然后计算可使用起始内存的页面号。
        end_mem -= start_mem;              // 再计算可分页处理的内存块大小。
        end_mem >>= 12;                    // 从而计算出可用于分页处理的页面数。
        while (end_mem-->0)                // 最后将这些可用页面对应的页面映射数组清零。
                mem_map[i++]=0;
}

// 计算内存空闲页面数并显示。
void calc_mem(void)
{
        int i,j,k,free=0;
        long * pg_tbl;

// 扫描内存页面映射数组 mem_map[],获取空闲页面数并显示。
        for(i=0 ; i<PAGING_PAGES ; i++)
                if (!mem_map[i]) free++;
        printk("%d pages free (of %d)\n\r",free,PAGING_PAGES);
// 扫描所有页目录项(除 0,1 项),如果页目录项有效,则统计对应页表中有效页面数,并显示。
        for(i=2 ; i<1024 ; i++) {
                if (1&pg_dir[i]) {
                        pg_tbl=(long *) (0xfffff000 & pg_dir[i]);
                        for(j=k=0 ; j<1024 ; j++)
                                if (pg_tbl[j]&1)
                                        k++;
                        printk("Pg-dir[%d] uses %d pages\n",i,k);
                }
        }
}
```

10.4 page.s 程序

10.4.1 功能描述

10.4.2 代码注释

列表 linux/mm/page.s 程序

```
/*
 *  linux/mm/page.s
 *
 *  (C) 1991  Linus Torvalds
 */

/*
```

```
 * page.s contains the low-level page-exception code.
 * the real work is done in mm.c
 */
/*
 * page.s 程序包含底层页异常处理代码。实际的工作在 memory.c 中完成。
 */

// 该文件包括页异常中断处理程序（中断 14），主要分两种情况处理。一是由于缺页引起的页异常中断，
// 通过调用 do_no_page(error_code, address)来处理；二是其它页保护引起的页异常，此时调用页写
// 保护处理函数 do_wp_page(error_code, address)进行处理。其中的出错码(error_code)是由 CPU 自
// 动产生并压入堆栈的，出现异常时访问的线性地址是从控制寄存器 CR2 中取得的。
.globl _page_fault

_page_fault:
        xchgl %eax,(%esp)       # 取出错码到 eax。
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%edx         # 置内核数据段选择符。
        mov %dx,%ds
        mov %dx,%es
        mov %dx,%fs
        movl %cr2,%edx          # 取引起页面异常的线性地址
        pushl %edx              # 将该线性地址和出错码压入堆栈，作为调用函数的参数。
        pushl %eax
        testl $1,%eax           # 测试标志 P，如果不是缺页引起的异常则跳转。
        jne 1f
        call _do_no_page        # 调用缺页处理函数（mm/memory.c,365 行）。
        jmp 2f
1:      call _do_wp_page        # 调用写保护处理函数（mm/memory.c,247 行）。
2:      addl $8,%esp            # 丢弃压入栈的两个参数。
        pop %fs
        pop %es
        pop %ds
        popl %edx
        popl %ecx
        popl %eax
        iret
```

10.4.3 其它信息

10.4.3.1 页异常的处理

当处理器在转换线性地址到物理地址的过程中检测到以下两种条件时，就会发生页异常中断，中断 14。

o 当 CPU 发现对应页目录项或页表项的存在位（Present）标志为 0。

o 当前进程没有访问指定页面的权限。

对于页异常处理中断，CPU 提供了两项信息用来诊断页异常和从中恢复运行。

(1) 放在堆栈上的出错码。该出错码指出了异常是由于页不存在引起的还是违反了访问权限引起的；在发生异常时 CPU 的当前特权层；以及是读操作还是写操作。出错码的格式是一个 32 位的长字。但只用了最后的 3 个比特位。分别说明导致异常发生时的原因：

位 2(U/S) – 0 表示在超级用户模式下执行，1 表示在用户模式下执行；
位 1(W/R) – 0 表示读操作，1 表示写操作；
位 0(P) – 0 表示页不存在，1 表示页级保护。

(2) CR2(控制寄存器 2)。CPU 将造成异常的用于访问的线性地址存放在 CR2 中。异常处理程序可以使用这个地址来定位相应的页目录和页表项。如果在页异常处理程序执行期间允许发生另一个页异常，那么处理程序应该将 CR2 压入堆栈中。

第 11 章 包含文件(include)

11.1 概述

Linux 0.11 版内核中共有 32 个头文件(*.h)，其中 asm/子目录中含有 4 个，linux/子目录中含有 10 个，sys/子目录中含有 5 个。从下一节开始我们首先描述 include/目录下的 13 个头文件，然后依次说明每个子目录中的文件。说明顺序按照文件名称排序进行。

11.2 include/目录下的文件

该目录下的文件列表如下（列表 11.1）：

列表 11.1 linux/include/目录下的文件

Name	Size	Last modified (GMT)	Description
asm/		1991-09-17 13:08:31	
linux/		1991-11-02 13:35:49	
sys/		1991-09-17 15:06:07	
a.out.h	6047 bytes	1991-09-17 15:10:49	m
const.h	321 bytes	1991-09-17 15:12:39	m
ctype.h	1049 bytes	1991-11-07 17:30:47	m
errno.h	1268 bytes	1991-09-17 15:04:15	m
fcntl.h	1374 bytes	1991-09-17 15:12:39	m
signal.h	1762 bytes	1991-09-22 19:58:04	m
stdarg.h	780 bytes	1991-09-17 15:02:23	m
stddef.h	286 bytes	1991-09-17 15:02:17	m
string.h	7881 bytes	1991-09-17 15:04:09	m
termios.h	5325 bytes	1991-11-25 20:02:08	m
time.h	734 bytes	1991-09-17 15:02:02	m
unistd.h	6410 bytes	1991-11-25 20:18:55	m
utime.h	225 bytes	1991-09-17 15:03:38	m

11.3 a.out.h 文件

11.3.1 功能描述

a.out.h 头文件主要定义了二进制执行文件 a.out(Assembley out)的格式。其中包括三个数据结构和一些宏函数。

11.3.2 代码注释

列表 11.2 linux/include/a.out.h 文件

```
#ifndef _A_OUT_H
#define _A_OUT_H

#define __GNU_EXEC_MACROS__

// 执行文件结构。
// =============================
// unsigned long a_magic        // 执行文件魔数。使用 N_MAGIC 等宏访问。
// unsigned a_text              // 代码长度，字节数。
// unsigned a_data              // 数据长度，字节数。
// unsigned a_bss               // 文件中的未初始化数据区长度，字节数。
// unsigned a_syms              // 文件中的符号表长度，字节数。
// unsigned a_entry             // 执行开始地址。
// unsigned a_trsize            // 代码重定位信息长度，字节数。
// unsigned a_drsize            // 数据重定位信息长度，字节数。
// -----------------------------
struct exec {
  unsigned long a_magic;        /* Use macros N_MAGIC, etc for access */
  unsigned a_text;              /* length of text, in bytes */
  unsigned a_data;              /* length of data, in bytes */
  unsigned a_bss;               /* length of uninitialized data area for file, in bytes */
  unsigned a_syms;              /* length of symbol table data in file, in bytes */
  unsigned a_entry;             /* start address */
  unsigned a_trsize;            /* length of relocation info for text, in bytes */
  unsigned a_drsize;            /* length of relocation info for data, in bytes */
};

// 用于取执行结构中的魔数。
#ifndef N_MAGIC
#define N_MAGIC(exec) ((exec).a_magic)
#endif

#ifndef OMAGIC
/* Code indicating object file or impure executable.  */
/* 指明为目标文件或者不纯的可执行文件的代号 */
#define OMAGIC 0407
/* Code indicating pure executable.  */
/* 指明为纯可执行文件的代号 */
#define NMAGIC 0410
/* Code indicating demand-paged executable.  */
/* 指明为需求分页处理的可执行文件 */
#define ZMAGIC 0413
#endif /* not OMAGIC */

// 如果魔数不能被识别，则返回真。
#ifndef N_BADMAG
#define N_BADMAG(x)                                   \
 (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC        \
  && N_MAGIC(x) != ZMAGIC)
#endif
```

```
#define _N_BADMAG(x)                                    \
 (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC          \
  && N_MAGIC(x) != ZMAGIC)

// 程序头在内存中的偏移位置。
#define _N_HDROFF(x) (SEGMENT_SIZE - sizeof (struct exec))

// 代码起始偏移值。
#ifndef N_TXTOFF
#define N_TXTOFF(x) \
 (N_MAGIC(x) == ZMAGIC ? _N_HDROFF((x)) + sizeof (struct exec) : sizeof (struct exec))
#endif

// 数据起始偏移值。
#ifndef N_DATOFF
#define N_DATOFF(x) (N_TXTOFF(x) + (x).a_text)
#endif

// 代码重定位信息偏移值。
#ifndef N_TRELOFF
#define N_TRELOFF(x) (N_DATOFF(x) + (x).a_data)
#endif

// 数据重定位信息偏移值。
#ifndef N_DRELOFF
#define N_DRELOFF(x) (N_TRELOFF(x) + (x).a_trsize)
#endif

// 符号表偏移值。
#ifndef N_SYMOFF
#define N_SYMOFF(x) (N_DRELOFF(x) + (x).a_drsize)
#endif

// 字符串信息偏移值。
#ifndef N_STROFF
#define N_STROFF(x) (N_SYMOFF(x) + (x).a_syms)
#endif

/* Address of text segment in memory after it is loaded.  */
/* 代码段加载到内存中后的地址 */
#ifndef N_TXTADDR
#define N_TXTADDR(x) 0
#endif

/* Address of data segment in memory after it is loaded.
   Note that it is up to you to define SEGMENT_SIZE
   on machines not listed here.  */
/* 数据段加载到内存中后的地址。
   注意，对于下面没有列出名称的机器，需要你自己来定义
   对应的 SEGMENT_SIZE */
#if defined(vax) || defined(hp300) || defined(pyr)
#define SEGMENT_SIZE PAGE_SIZE
```

```
#endif
#ifdef  hp300
#define PAGE_SIZE       4096
#endif
#ifdef  sony
#define SEGMENT_SIZE    0x2000
#endif  /* Sony.  */
#ifdef is68k
#define SEGMENT_SIZE 0x20000
#endif
#if defined(m68k) && defined(PORTAR)
#define PAGE_SIZE 0x400
#define SEGMENT_SIZE PAGE_SIZE
#endif

#define PAGE_SIZE 4096
#define SEGMENT_SIZE 1024

// 以段为界的大小。
#define _N_SEGMENT_ROUND(x) (((x) + SEGMENT_SIZE - 1) & ~(SEGMENT_SIZE - 1))

// 代码段尾地址。
#define _N_TXTENDADDR(x) (N_TXTADDR(x)+(x).a_text)

// 数据开始地址。
#ifndef N_DATADDR
#define N_DATADDR(x) \
    (N_MAGIC(x)==OMAGIC? (_N_TXTENDADDR(x)) \
     : (_N_SEGMENT_ROUND (_N_TXTENDADDR(x))))
#endif

/* Address of bss segment in memory after it is loaded.  */
/* bss 段加载到内存以后的地址 */
#ifndef N_BSSADDR
#define N_BSSADDR(x) (N_DATADDR(x) + (x).a_data)
#endif

// nlist 结构。
#ifndef N_NLIST_DECLARED
struct nlist {
  union {
    char *n_name;
    struct nlist *n_next;
    long n_strx;
  } n_un;
  unsigned char n_type;
  char n_other;
  short n_desc;
  unsigned long n_value;
};
#endif

// 下面定义 exec 结构中的变量偏移值。
```

```
#ifndef N_UNDF
#define N_UNDF 0
#endif
#ifndef N_ABS
#define N_ABS 2
#endif
#ifndef N_TEXT
#define N_TEXT 4
#endif
#ifndef N_DATA
#define N_DATA 6
#endif
#ifndef N_BSS
#define N_BSS 8
#endif
#ifndef N_COMM
#define N_COMM 18
#endif
#ifndef N_FN
#define N_FN 15
#endif

#ifndef N_EXT
#define N_EXT 1
#endif
#ifndef N_TYPE
#define N_TYPE 036
#endif
#ifndef N_STAB
#define N_STAB 0340
#endif

/* The following type indicates the definition of a symbol as being
   an indirect reference to another symbol.  The other symbol
   appears as an undefined reference, immediately following this symbol.

   Indirection is asymmetrical.  The other symbol's value will be used
   to satisfy requests for the indirect symbol, but not vice versa.
   If the other symbol does not have a definition, libraries will
   be searched to find a definition.  */
/* 下面的类型指明了符号的定义作为对另一个符号的间接引用。紧接该符号的其它
 * 的符号呈现为未定义的引用。
 *
 * 间接性是不对称的。其它符号的值将被用于满足间接符号的请求，但反之不然。
 * 如果其它符号并没有定义，则将搜索库来寻找一个定义 */
#define N_INDR 0xa

/* The following symbols refer to set elements.
   All the N_SET[ATDB] symbols with the same name form one set.
   Space is allocated for the set in the text section, and each set
   element's value is stored into one word of the space.
   The first word of the space is the length of the set (number of elements).

```

```
   The address of the set is made into an N_SETV symbol
   whose name is the same as the name of the set.
   This symbol acts like a N_DATA global symbol
   in that it can satisfy undefined external references.  */
/* 下面的符号与集合元素有关。所有具有相同名称 N_SET[ATDB]的符号
   形成一个集合。在代码部分中已为集合分配了空间，并且每个集合元素
   的值存放在一个字（word）的空间。空间的第一个字存有集合的长度（集合元素数目）。

   集合的地址被放入一个 N_SETV 符号，它的名称与集合同名。
   在满足未定义的外部引用方面，该符号的行为象一个 N_DATA 全局符号。*/

/* These appear as input to LD, in a .o file.  */
/* 以下这些符号在目标文件中是作为链接程序 LD 的输入。*/
#define N_SETA  0x14            /* Absolute set element symbol */
                                /* 绝对集合元素符号 */
#define N_SETT  0x16            /* Text set element symbol */
                                /* 代码集合元素符号 */
#define N_SETD  0x18            /* Data set element symbol */
                                /* 数据集合元素符号 */
#define N_SETB  0x1A            /* Bss set element symbol */
                                /* Bss 集合元素符号 */

/* This is output from LD.  */
/* 下面是 LD 的输出。*/
#define N_SETV  0x1C            /* Pointer to set vector in data area.  */
                                /* 指向数据区中集合向量。*/

#ifndef N_RELOCATION_INFO_DECLARED

/* This structure describes a single relocation to be performed.
   The text-relocation section of the file is a vector of these structures,
   all of which apply to the text section.
   Likewise, the data-relocation section applies to the data section.  */
/* 下面的结构描述执行一个重定位的操作。
   文件的代码重定位部分是这些结构的一个向量，所有这些适用于代码部分。
   类似地，数据重定位部分适用于数据部分。*/

// 重定位信息结构。
struct relocation_info
{
  /* Address (within segment) to be relocated.  */
  /* 需要重定位的地址（在段内）。*/
  int r_address;
  /* The meaning of r_symbolnum depends on r_extern.  */
  /* r_symbolnum 的含义与 r_extern 有关。*/
  unsigned int r_symbolnum:24;
  /* Nonzero means value is a pc-relative offset
     and it should be relocated for changes in its own address
     as well as for changes in the symbol or section specified.  */
  /* 非零意味着值是一个 pc 相关的偏移值，因而需要被重定位到自己
     的地址处以及符号或节指定的改变。 */ [??]
  unsigned int r_pcrel:1;
  /* Length (as exponent of 2) of the field to be relocated.
```

```
        Thus, a value of 2 indicates 1<<2 bytes.  */
    /* 需要被重定位的字段长度（是 2 的次方）。
       因此，若值是 2 则表示 1<<2 字节数。*/
    unsigned int r_length:2;
    /* 1 => relocate with value of symbol.
             r_symbolnum is the index of the symbol
             in file's the symbol table.
       0 => relocate with the address of a segment.
             r_symbolnum is N_TEXT, N_DATA, N_BSS or N_ABS
             (the N_EXT bit may be set also, but signifies nothing).  */
    /* 1 => 以符号的值重定位。
             r_symbolnum 是文件符号表中符号的索引。
       0 => 以段的地址进行重定位。
             r_symbolnum 是 N_TEXT、N_DATA、N_BSS 或 N_ABS
             (N_EXT 比特位也可以被设置，但是毫无意义)。*/
    unsigned int r_extern:1;
    /* Four bits that aren't used, but when writing an object file
       it is desirable to clear them.  */
    /* 没有使用的 4 个比特位，但是当进行写一个目标文件时
       最好将它们复位掉。*/
    unsigned int r_pad:4;
  };
  #endif /* no N_RELOCATION_INFO_DECLARED.  */


  #endif /* __A_OUT_GNU_H__ */
```

11.3.3 其它信息

a.out 执行文件格式

Linux 内核 0.11 版仅支持 a.out(Assembley out)执行文件格式，虽然这种格式目前已经渐渐不用，而使用功能更为齐全的 ELF（Executable and Link Format）格式，但是由于其简单性，作为学习入门的材料正好比较适用。下面全面介绍一下 a.out 格式。

在头文件<a.out.h>中申明了三个数据结构以及一些宏函数。这些数据结构描述了系统上可执行的机器码文件（二进制文件）。

一个执行文件共可有七个部分（七节）组成。按照顺序，这些部分是：

执行头部分(exec header)

执行文件头部分。该部分中含有一些参数，内核使用这些参数将执行文件加载到内存中并执行，而链接程序(ld)使用这些参数将一些二进制目标文件组合成一个可执行文件。这是唯一必要的组成部分。

代码段部分(text segment)

含有程序执行使被加载到内存中的指令代码和相关数据。可以以只读形式进行加载。

数据段部分(data segment)

这部分含有已经初始化过的数据，总是被加载到可读写的内存中。

代码重定位部分(text relocations)
这部分含有供链接程序使用的记录数据。在组合二进制目标文件时用于定位代码段中的指针或地址。

数据重定位部分(data relocations)
与代码重定位部分的作用类似，但是是用于数据段中指针的重定位。

符号表部分(simbol table)
这部分同样含有供链接程序使用的记录数据，用于在二进制目标文件之间对命名的变量和函数（符号）进行交叉引用。

字符串表部分(string table)
该部分含有与符号名相对应的字符串。

每个二进制执行文件均以一个执行数据结构（exec structure）开始。该数据结构的形式如下：

```
struct exec {
        unsigned long a_midmag;
        unsigned long a_text;
        unsigned long a_data;
        unsigned long a_bss;
        unsigned long a_syms;
        unsigned long a_entry;
        unsigned long a_trsize;
        unsigned long a_drsize;
};
```

各个字段的功能如下：

a_midmag　该字段含有被 N_GETFLAG()、N_GETMID 和 N_GETMAGIC()访问的子部分，是由链接程序
在　运行时加载到进程地址空间。宏 N_GETMID()用于返回机器标识符(machine-id)，指示
出　二进制文件将在什么机器上运行。N_GETMAGIC()宏指明魔数，它唯一地确定了二进制
执　行文件与其它加载的文件之间的区别。字段中必须包含以下值之一：

OMAGIC　表示代码和数据段紧随在执行头后面并且是连续存放的。内核将代码和数据
段　都加载到可读写内存中。

NMAGIC　同 OMAGIC 一样，代码和数据段紧随在执行头后面并且是连续存放的。然而内核将代码加载到了只读内存中，并把数据段加载到代码段后下一页可读
写内存　边界开始。

ZMAGIC　内核在必要时从二进制执行文件中加载独立的页面。执行头部、代码段和数
据　段都被链接程序处理成多个页面大小的块。内核加载的代码页面时只读的，
而　数据段的页面是可写的。

a_text　该字段含有代码段的长度值，字节数。

a_data　该字段含有数据段的长度值，字节数。

a_bss   含有‘bss 段’的长度，内核用其设置在数据段后初始的 break（brk）。内核在加载程序时，这段可写内存显现出处于数据段后面，并且初始时为全零。

a_syms   含有符号表部分的字节长度值。

a_entry   含有内核将执行文件加载到内存中以后，程序执行起始点的内存地址。

a_trsize   该字段含有代码重定位表的大小，是字节数。

a_drsize   该字段含有数据重定位表的大小，是字节数。

在 a.out.h 头文件中定义了几个宏，这些宏使用 exec 结构来测试一致性或者定位执行文件中各个部分（节）的位置偏移值。这些宏有：

N_BADMAG(exec) 如果 a_magic 字段不能被识别，则返回非零值。

N_TXTOFF(exec) 代码段的起始位置字节偏移值。

N_DATOFF(exec) 数据段的起始位置字节偏移值。

N_DRELOFF(exec) 数据重定位表的起始位置字节偏移值。

N_TRELOFF(exec) 代码重定位表的起始位置字节偏移值。

N_SYMOFF(exec) 符号表的起始位置字节偏移值。

N_STROFF(exec) 字符串表的起始位置字节偏移值。

重定位记录具有标准格式，它使用重定位信息(relocation_info)结构来描述：

```
struct relocation_info {
        int             r_address;
        unsigned int    r_symbolnum : 24,
                        r_pcrel : 1,
                        r_length : 2,
                        r_extern : 1,
                        r_baserel : 1,
                        r_jmptable : 1,
                        r_relative : 1,
                        r_copy : 1;
};
```

该结构中各字段的含义如下：

r_address 该字段含有需要链接程序处理（编辑）的指针的字节偏移值。代码重定位的偏移值是从代码段开始处计数的，数据重定位的偏移值是从数据段开始处计算的。链接程序会将已经存储在该偏移处的值与使用重定位记录计算出的新值相加。

r_symbolnum 该字段含有符号表中一个符号结构的序号值（不是字节偏移值）。链接程序在算出符号的绝对地址以后，就将该地址加到正在进行重定位的指针上。（如果 r_extern 比特位是 0，那么情况就不同，见下面。）

r_pcrel 如果设置了该位，链接程序就认为正在更新一个指针，该指针使用 pc 相关寻址方式，是属于机器码指令部分。当运行程序使用这个被重定位的指针时，该指针的地址被隐式地加到该指针上。

r_length 该字段含有指针长度的 2 的次方值：0 表示 1 字节长，1 表示 2 字节长，2 表示 4 字节长。

r_extern 如果被置位，表示该重定位需要一个外部引用；此时链接程序必须使用一个符号地址来更新相应指针。当该位是 0 时，则重定位是“局部”的；链接程序更新指针以反映各个段加载地址中的变化，而不是反映一个符号值的变化（除非同时设置了 r_baserel，见下面)。在这种情况下，r_symbolnum 字段的内容是一个 n_type 值（见下面)；这类字段告诉链接程序被重定位的指针指向那个段。

r_baserel 如果设置了该位，则 r_symbolnum 字段指定的符号将被重定位成全局偏移表(Global Offset Table)中的一个偏移值。在运行时刻，全局偏移表该偏移处被设置为符号的地址。

r_jmptable 如果被置位，则 r_symbolnum 字段指定的符号将被重定位成过程链接表(Procedure Linkage Table)中的一个偏移值。

r_relative 如果被置位，则说明此重定位与该目标文件将成为其组成部分的映象文件在运行时被加载的地址相关。这类重定位仅在共享目标文件中出现。

r_copy 如果被置位，该重定位记录指定了一个符号，该符号的内容将被复制到 r_address 指定的地方。该复制操作是通过共享目标模块中一个合适的数据项中的运行时刻链接程序完成的。

符号将名称映射为地址（或者更通俗地讲是字符串映射到值)。由于链接程序对地址的调整，一个符号的名称必须用来表示其地址，直到已被赋予一个绝对地址值。符号是由符号表中固定长度的记录以及字符串表中的可变长度名称组成。符号表是 nlist 结构的一个数组，如下所示：

```
struct nlist {
        union {
                char    *n_name;
                long    n_strx;
        } n_un;
        unsigned char   n_type;
        char            n_other;
```

```
        short           n_desc;
        unsigned long   n_value;
};
```

其中各字段的含义为：

n_un.n_strx 含有本符号的名称在字符串表中的字节偏移值。当程序使用 nlist()函数访问一个符号表时，该字段被替换为 n_un.n_name 字段，这是内存中字符串的指针。

n_type 用于链接程序确定如何更新符号的值。使用位屏蔽(bitmasks)可以将 n_type 字段分割成三个子字段，对于 N_EXT 类型位置位的符号，链接程序将它们看作是“外部的”符号，并且允许其它二进制目标文件对它们的引用。N_TYPE 屏蔽码用于链接程序感兴趣的比特位：

N_UNDF 一个未定义的符号。链接程序必须在其它二进制目标文件中定位一个具有相
同 名称的外部符号，以确定该符号的绝对数据值。特殊情况下，如果 n_type 字段是非零
值，并且没有二进制文件定义了这个符号，则链接程序在 BSS 段中将该符号解析
为一个 地址，保留长度等于 n_value 的字节。如果符号在多于一个二进制目标文件中都
没有定 义并且这些二进制目标文件对其长度值不一致，则链接程序将选择所有二进制目
标文件 中最大的长度。

N_ABS 一个绝对符号。链接程序不会更新一个绝对符号。

N_TEXT 一个代码符号。该符号的值是代码地址，链接程序在合并二进制目标文件时
会 更新其值。

N_DATA 一个数据符号；与 N_TEXT 类似，但是用于数据地址。对应代码和数据符号的
值不是文件的偏移值而是地址；为了找出文件的偏移，就有必要确定相关部分开
始加载 的地址并减去它，然后加上该部分的偏移。

N_BSS 一个 BSS 符号；与代码或数据符号类似，但在二进制目标文件中没有对应的
偏 移。

N_FN 一个文件名符号。在合并二进制目标文件时，链接程序会将该符号插入在二
进 制文件中的符号之前。符号的名称就是给予链接程序的文件名，而其值是二进制文件
中 首个代码段地址。链接和加载时不需要文件名符号，但对于调式程序非常有用。

N_STAB 屏蔽码用于选择符号调式程序(例如 gdb)感兴趣的位；其值在 stab()中说明。

n_other 该字段按照 n_type 确定的段，提供有关符号重定位操作的符号独立性信息。目前，n_other 字段的最低 4 位含有两个值之一：AUX_FUNC 和 AUX_OBJECT（有关定义参见<link.h>）。AUX_FUNC 将符号与可调用的函数相关，AUX_OBJECT 将符号与数据相关，而不管它们是位于代码段还是数据段。该字段主要用于链接程序 ld，用于动态可执行程序的创建。

n_desc 保留给调式程序使用；链接程序不对其进行处理。不同的调试程序将该字段用作不同

的用途。

n_value 含有符号的值。对于代码、数据和 BSS 符号，这是一个地址；对于其它符号（例如调式程序符号），值可以是任意的。

字符串表是由长度为 u_int32_t 后跟一 null 结尾的符号字符串组成。长度代表整个表的字节大小，所以在 32 位的机器上其最小值（或者是第 1 个字符串的偏移）总是 4。

11.4 const.h 文件

11.4.1 功能描述

该文件中定义了 i 节点中文件属性和类型 i_mode 字段所用到的一些标志位常量符号。

11.4.2 代码注释

列表 linux/include/const.h 文件

```
#ifndef _CONST_H
#define _CONST_H

#define BUFFER_END 0x200000          // 定义缓冲使用内存的末端(代码中没有使用该常量)。

// i 节点数据结构中 i_mode 字段的各标志位。
#define I_TYPE          0170000      // 指明 i 节点类型。
#define I_DIRECTORY     0040000      // 是目录文件。
#define I_REGULAR       0100000      // 常规文件，不是目录文件或特殊文件。
#define I_BLOCK_SPECIAL 0060000      // 块设备特殊文件。
#define I_CHAR_SPECIAL  0020000      // 字符设备特殊文件。
#define I_NAMED_PIPE    0010000      // 命名管道。
#define I_SET_UID_BIT   0004000      // 在执行时设置有效用户 id 类型。
#define I_SET_GID_BIT   0002000      // 在执行时设置有效组 id 类型。

#endif

```

11.5 ctype.h 文件

11.5.1 功能描述

该文件定义了一些有关字符类型判断和转换的宏，是使用数组（表）进行操作的。当使用宏时，字符是作为一个表(__ctype)中的索引，从表中获取一个字节，于是可得到相关的比特位。

11.5.2 代码注释

列表 linux/include/ctype.h 文件

```
#ifndef _CTYPE_H
#define _CTYPE_H

#define _U      0x01    /* upper */          // 该比特位用于大写字符[A-Z]。
#define _L      0x02    /* lower */          // 该比特位用于小写字符[a-z]。
#define _D      0x04    /* digit */          // 该比特位用于数字[0-9]。
#define _C      0x08    /* cntrl */          // 该比特位用于控制字符。
#define _P      0x10    /* punct */          // 该比特位用于标点字符。
#define _S      0x20    /* white space (space/lf/tab) */// 用于空白字符，如空格、\t、\n 等。
#define _X      0x40    /* hex digit */      // 该比特位用于十六进制数字。
#define _SP     0x80    /* hard space (0x20) */   // 该比特位用于空格字符(0x20)。

extern unsigned char _ctype[];          // 字符特性数组(表)，定义了各个字符对应上面的属性。
extern char _ctmp;                      // 一个临时字符变量(在 fs/ctype.c 中定义)。

// 下面是一些确定字符类型的宏。
#define isalnum(c) ((_ctype+1)[c]&(_U|_L|_D))         // 是字符或数字[A-Z]、[a-z]或[0-9]。
#define isalpha(c) ((_ctype+1)[c]&(_U|_L))            // 是字符。
#define iscntrl(c) ((_ctype+1)[c]&(_C))               // 是控制字符。
#define isdigit(c) ((_ctype+1)[c]&(_D))               // 是数字。
#define isgraph(c) ((_ctype+1)[c]&(_P|_U|_L|_D))      // 是图形字符。
#define islower(c) ((_ctype+1)[c]&(_L))               // 是小写字符。
#define isprint(c) ((_ctype+1)[c]&(_P|_U|_L|_D|_SP))  // 是可打印字符。
#define ispunct(c) ((_ctype+1)[c]&(_P))               // 是标点符号。
#define isspace(c) ((_ctype+1)[c]&(_S))               // 是空白字符如空格,\f,\n,\r,\t,\v。
#define isupper(c) ((_ctype+1)[c]&(_U))               // 是大写字符。
#define isxdigit(c) ((_ctype+1)[c]&(_D|_X))           // 是十六进制数字。

#define isascii(c) (((unsigned) c)<=0x7f)             // 是 ASCII 字符。
#define toascii(c) (((unsigned) c)&0x7f)              // 转换成 ASCII 字符。

#define tolower(c) (_ctmp=c,isupper(_ctmp)?_ctmp-('A'-'a'):_ctmp)  // 转换成对应小写字符。
#define toupper(c) (_ctmp=c,islower(_ctmp)?_ctmp-('a'-'A'):_ctmp)  // 转换成对应大写字符。

#endif
```

11.6 errno.h 文件

11.6.1 功能描述

11.6.2 代码注释

列表 linux/include/errno.h 文件

```
#ifndef _ERRNO_H
```

```
#define _ERRNO_H

/*
 * ok, as I hadn't got any other source of information about
 * possible error numbers, I was forced to use the same numbers
 * as minix.
 * Hopefully these are posix or something. I wouldn't know (and posix
 * isn't telling me - they want $$$ for their f***ing standard).
 *
 * We don't use the _SIGN cludge of minix, so kernel returns must
 * see to the sign by themselves.
 *
 * NOTE! Remember to change strerror() if you change this file!
 */
/*
 * ok，由于我没有得到任何其它有关出错号的资料，我只能使用与 minix 系统
 * 相同的出错号了。
 * 希望这些是 POSIX 兼容的或者在一定程度上是这样的，我不知道（而且 POSIX
 * 没有告诉我 - 要获得他们的混蛋标准需要出钱）。
 *
 * 我们没有使用 minix 那样的_SIGN 簇，所以内核的返回值必须自己辨别正负号。
 *
 * 注意！如果你改变该文件的话，记着也要修改 strerror()函数。
 */

extern int errno;

#define ERROR           99              // 一般错误。
#define EPERM            1              // 操作没有许可。
#define ENOENT           2              // 文件或目录不存在。
#define ESRCH            3              // 指定的进程不存在。
#define EINTR            4              // 中断的函数调用。
#define EIO              5              // 输入/输出错。
#define ENXIO            6              // 指定设备或地址不存在。
#define E2BIG            7              // 参数列表太长。
#define ENOEXEC          8              // 执行程序格式错误。
#define EBADF            9              // 文件句柄(描述符)错误。
#define ECHILD          10              // 子进程不存在。
#define EAGAIN          11              // 资源暂时不可用。
#define ENOMEM          12              // 内存不足。
#define EACCES          13              // 没有许可权限。
#define EFAULT          14              // 地址错。
#define ENOTBLK         15              // 不是块设备文件。
#define EBUSY           16              // 资源正忙。
#define EEXIST          17              // 文件已存在。
#define EXDEV           18              // 非法连接。
#define ENODEV          19              // 设备不存在。
#define ENOTDIR         20              // 不是目录文件。
#define EISDIR          21              // 是目录文件。
#define EINVAL          22              // 参数无效。
#define ENFILE          23              // 系统打开文件数太多。
#define EMFILE          24              // 打开文件数太多。
#define ENOTTY          25              // 不恰当的 IO 控制操作(没有 tty 终端)。
```

```
#define ETXTBSY         26                  // 不再使用。
#define EFBIG           27                  // 文件太大。
#define ENOSPC          28                  // 设备已满（设备已经没有空间）。
#define ESPIPE          29                  // 无效的文件指针重定位。
#define EROFS           30                  // 文件系统只读。
#define EMLINK          31                  // 连接太多。
#define EPIPE           32                  // 管道错。
#define EDOM            33                  // 域(domain)出错。
#define ERANGE          34                  // 结果太大。
#define EDEADLK         35                  // 避免资源死锁。
#define ENAMETOOLONG    36                  // 文件名太长。
#define ENOLCK          37                  // 没有锁定可用。
#define ENOSYS          38                  // 功能还没有实现。
#define ENOTEMPTY       39                  // 目录不空。

#endif

```

11.7 fcntl.h 文件

11.10.1 功能描述

11.7.2 代码注释

列表 linux/include/fcntl.h 文件

```
#ifndef _FCNTL_H
#define _FCNTL_H

#include <sys/types.h>     // 类型头文件。定义了基本的系统数据类型。

/* open/fcntl - NOCTTY, NDELAY isn't implemented yet */
/* open/fcntl - NOCTTY 和 NDELAY 现在还没有实现 */
#define O_ACCMODE       00003                   // 文件访问模式屏蔽码。
// 打开文件 open()和文件控制 fcntl()函数使用的文件访问模式。同时只能使用三者之一。
#define O_RDONLY           00                   // 以只读方式打开文件。
#define O_WRONLY           01                   // 以只写方式打开文件。
#define O_RDWR             02                   // 以读写方式打开文件。
// 下面是文件创建标志，用于 open()。可与上面访问模式用'位或'的方式一起使用。
#define O_CREAT         00100   /* not fcntl */ // 如果文件不存在就创建。
#define O_EXCL          00200   /* not fcntl */ // 独占使用文件标志。
#define O_NOCTTY        00400   /* not fcntl */ // 不分配控制终端。
#define O_TRUNC         01000   /* not fcntl */ // 若文件已存在且是写操作，则长度截为 0。
#define O_APPEND        02000                   // 以添加方式打开，文件指针置为文件尾。
#define O_NONBLOCK      04000   /* not fcntl */ // 非阻塞方式打开和操作文件。
#define O_NDELAY        O_NONBLOCK              // 非阻塞方式打开和操作文件。

/* Defines for fcntl-commands. Note that currently
 * locking isn't supported, and other things aren't really
```

```
21  * tested.
22  */
   /* 下面定义了 fcntl 的命令。注意目前锁定命令还没有支持，而其它
    * 命令实际上还没有测试过。
    */
   // 文件句柄(描述符)操作函数 fcntl()的命令。
23 #define F_DUPFD        0       /* dup */           // 拷贝文件句柄为最小数值的句柄。
24 #define F_GETFD        1       /* get f_flags */   // 取文件句柄标志。
25 #define F_SETFD        2       /* set f_flags */   // 设置文件句柄标志。
26 #define F_GETFL        3       /* more flags (cloexec) */  // 取文件状态标志和访问模式。
27 #define F_SETFL        4                                   // 设置文件状态标志和访问模式。
   // 下面是文件锁定命令。fcntl()的第三个参数 lock 是指向 flock 结构的指针。
28 #define F_GETLK        5       /* not implemented */  // 返回阻止锁定的 flock 结构。
29 #define F_SETLK        6       // 设置(F_RDLCK 或 F_WRLCK)或清除(F_UNLCK)锁定。
30 #define F_SETLKW       7       // 等待设置或清除锁定。
31
32 /* for F_[GET|SET]FL */
   /* 用于 F_GETFL 或 F_SETFL */
   // 在执行 exec()簇函数时关闭文件句柄。(执行时关闭 - Close On EXECution)
33 #define FD_CLOEXEC     1       /* actually anything with low bit set goes */
                               /* 实际上只要低位为 1 即可 */
34
35 /* Ok, these are locking features, and aren't implemented at any
36  * level. POSIX wants them.
37  */
   /* OK，以下是锁定类型，任何函数中都还没有实现。POSIX 标准要求这些类型。
    */
38 #define F_RDLCK        0       // 共享或读文件锁定。
39 #define F_WRLCK        1       // 独占或写文件锁定。
40 #define F_UNLCK        2       // 文件解锁。
41
42 /* Once again - not implemented, but ... */
   /* 同样 - 也还没有实现，但是... */
   // 文件锁定操作数据结构。描述了受影响文件段的类型(l_type)、开始偏移(l_whence)、
   // 相对偏移(l_start)、锁定长度(l_len)和实施锁定的进程 id。
43 struct flock {
44         short l_type;           // 锁定类型（F_RDLCK，F_WRLCK，F_UNLCK）。
45         short l_whence;         // 开始偏移(SEEK_SET，SEEK_CUR 或 SEEK_END)。
46         off_t l_start;          // 阻塞锁定的开始处。相对偏移（字节数）。
47         off_t l_len;            // 阻塞锁定的大小；如果是 0 则为到文件末尾。
48         pid_t l_pid;            // 加锁的进程 id。
49 };
50
   // 以下是使用上述标志或命令的函数原型。
   // 创建新文件或重写一个已存在文件。
   // 参数 filename 是欲创建文件的文件名，mode 是创建文件的属性（参见 include/sys/stat.h）。
51 extern int creat(const char * filename,mode_t mode);
   // 文件句柄操作，会影响文件的打开。
   // 参数 fildes 是文件句柄，cmd 是操作命令，见上面 23-30 行。
52 extern int fcntl(int fildes,int cmd, ...);
   // 打开文件。在文件与文件句柄之间建立联系。
   // 参数 filename 是欲打开文件的文件名，flags 是上面 7-17 行上的标志的组合。
53 extern int open(const char * filename, int flags, ...);
```

```

#endif

```

11.8 signal.h 文件

11.8.1 功能描述

11.8.2 文件注释

列表 linux/include/signal.h 文件

```
#ifndef _SIGNAL_H
#define _SIGNAL_H

#include <sys/types.h>                  // 类型头文件。定义了基本的系统数据类型。

typedef int sig_atomic_t;               // 定义信号原子操作类型。
typedef unsigned int sigset_t;          /* 32 bits */  // 定义信号集类型。

#define _NSIG             32            // 定义信号种类 -- 32 种。
#define NSIG            _NSIG           // NSIG = _NSIG

// 以下这些是 Linux 0.11 内核中定义的信号。
#define SIGHUP             1            // Hang Up    -- 挂断控制终端或进程。
#define SIGINT             2            // Interrupt  -- 来自键盘的中断。
#define SIGQUIT            3            // Quit       -- 来自键盘的退出。
#define SIGILL             4            // Illeagle   -- 非法指令。
#define SIGTRAP            5            // Trap       -- 跟踪断点。
#define SIGABRT            6            // Abort      -- 异常结束。
#define SIGIOT             6            // IO Trap    -- 同上。
#define SIGUNUSED          7            // Unused     -- 没有使用。
#define SIGFPE             8            // FPE        -- 协处理器出错。
#define SIGKILL            9            // Kill       -- 强迫进程终止。
#define SIGUSR1           10            // User1      -- 用户信号 1，进程可使用。
#define SIGSEGV           11            // Segment Violation -- 无效内存引用。
#define SIGUSR2           12            // User2      -- 用户信号 2，进程可使用。
#define SIGPIPE           13            // Pipe       -- 管道写出错，无读者。
#define SIGALRM           14            // Alarm      -- 实时定时器报警。
#define SIGTERM           15            // Terminate  -- 进程终止。
#define SIGSTKFLT         16            // Stack Fault -- 栈出错（协处理器）。
#define SIGCHLD           17            // Child      -- 子进程停止或被终止。
#define SIGCONT           18            // Continue   -- 恢复进程继续执行。
#define SIGSTOP           19            // Stop       -- 停止进程的执行。
#define SIGTSTP           20            // TTY Stop   -- tty 发出停止进程，可忽略。
#define SIGTTIN           21            // TTY In     -- 后台进程请求输入。
#define SIGTTOU           22            // TTY Out    -- 后台进程请求输出。

/* Ok, I haven't implemented sigactions, but trying to keep headers POSIX */
```

```
   /* OK，我还没有实现 sigactions 的编制，但在头文件中仍希望遵守 POSIX 标准 */
37 #define SA_NOCLDSTOP    1           // 当子进程处于停止状态，就不对 SIGCHLD 处理。
38 #define SA_NOMASK       0x40000000  // 不阻止在指定的信号处理程序(信号句柄)中再收到该信号。
39 #define SA_ONESHOT      0x80000000  // 信号句柄一旦被调用过就恢复到默认处理句柄。
40
   // 以下参数用于 sigprocmask()-- 改变阻塞信号集(屏蔽码)。这些参数可以改变该函数的行为。
41 #define SIG_BLOCK          0    /* for blocking signals */
                                // 在阻塞信号集中加上给定的信号集。
42 #define SIG_UNBLOCK        1    /* for unblocking signals */
                                // 从阻塞信号集中删除指定的信号集。
43 #define SIG_SETMASK        2    /* for setting the signal mask */
                                // 设置阻塞信号集（信号屏蔽码）。
44
45 #define SIG_DFL         ((void (*)(int))0)      /* default signal handling */
                                                // 默认的信号处理程序（信号句柄）。
46 #define SIG_IGN         ((void (*)(int))1)      /* ignore signal */
                                                // 忽略信号的处理程序。
47
   // 下面是 sigaction 的数据结构。
   // sa_handler 是对应某信号指定要采取的行动。可以是上面的 SIG_DFL，或者是 SIG_IGN 来忽略
   // 该信号，也可以是指向处理该信号函数的一个指针。
   // sa_mask 给出了对信号的屏蔽码，在信号程序执行时将阻塞对这些信号的处理。
   // sa_flags 指定改变信号处理过程的信号集。它是由 37-39 行的位标志定义的。
   // sa_restorer 恢复过程指针，是用于保存原返回的过程指针。
   // 另外，引起触发信号处理的信号也将被阻塞，除非使用了 SA_NOMASK 标志。
48 struct sigaction {
49         void (*sa_handler)(int);
50         sigset_t sa_mask;
51         int sa_flags;
52         void (*sa_restorer)(void);
53 };
54
   // 为信号_sig 安装一个新的信号处理程序（信号句柄），与 sigaction()类似。
55 void (*signal(int _sig, void (*_func)(int)))(int);
   // 向当前进程发送一个信号。其作用等价于 kill(getpid(),sig)。
56 int raise(int sig);
   // 可用于向任何进程组或进程发送任何信号。
57 int kill(pid_t pid, int sig);
   // 向信号集中添加信号。
58 int sigaddset(sigset_t *mask, int signo);
   // 从信号集中去除指定的信号。
59 int sigdelset(sigset_t *mask, int signo);
   // 从信号集中清除指定信号集。
60 int sigemptyset(sigset_t *mask);
   // 向信号集中置入所有信号。
61 int sigfillset(sigset_t *mask);
   // 判断一个信号是否是信号集中的。1 -- 是， 0 -- 不是， -1 -- 出错。
62 int sigismember(sigset_t *mask, int signo); /* 1 - is, 0 - not, -1 error */
   // 对 set 中的信号进行检测，看是否有挂起的信号。
63 int sigpending(sigset_t *set);
   // 改变目前的被阻塞信号集（信号屏蔽码）。
64 int sigprocmask(int how, sigset_t *set, sigset_t *oldset);
   // 用 sigmask 临时替换进程的信号屏蔽码,然后暂停该进程直到收到一个信号。
```

```
int sigsuspend(sigset_t *sigmask);
// 用于改变进程在收到指定信号时所采取的行动。
int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);

#endif /* _SIGNAL_H */

```

11.9 stdarg.h 文件

11.9.1 功能描述

stdarg.h 是标准参数头文件。它以宏的形式定义变量参数列表。主要说明了-个类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于 vsprintf、vprintf、vfprintf 函数。

11.9.2 代码注释

列表 linux/include/stdarg.h 文件

```
#ifndef _STDARG_H
#define _STDARG_H

typedef char *va_list;  // 定义 va_list 是一个字符指针类型。

/* Amount of space required in an argument list for an arg of type TYPE.
   TYPE may alternatively be an expression whose type is used.  */
/* 下面给出了类型为 TYPE 的 arg 参数列表所要求的空间容量。
   TYPE 也可以是使用该类型的一个表达式 */

// 下面这句定义了取整后的 TYPE 类型的字节长度值。是 int 长度(4)的倍数。
#define __va_rounded_size(TYPE)  \
  (((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))

// 下面这个函数（用宏实现）使 AP 指向传给函数的可变参数表的第一个参数。
// 在第一次调用 va_arg 或 va_end 之前，必须首先调用该函数。
#ifndef __sparc__
#define va_start(AP, LASTARG)                                          \
 (AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
#else
#define va_start(AP, LASTARG)                                          \
 (__builtin_saveregs (),                                               \
  AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
#endif

// 下面该宏用于被调用函数完成一次正常返回。va_end 可以修改 AP 使其在重新调用
// va_start 之前不能被使用。va_end 必须在 va_arg 读完所有的参数后再被调用。
void va_end (va_list);          /* Defined in gnulib */  /* 在 gnulib 中定义 */
#define va_end(AP)

// 下面该宏用于扩展表达式使其与下一个被传递参数具有相同的类型和值。
// 对于缺省值，va_arg 可以用字符、无符号字符和浮点类型。
// 在第一次使用 va_arg 时，它返回表中的第一个参数，后续的每次调用都将返回表中的
```

```
// 下一个参数。这是通过先访问 AP，然后把它增加以指向下一项来实现的。
// va_arg 使用 TYPE 来完成访问和定位下一项，每调用一次 va_arg，它就修改 AP 以指示
// 表中的下一参数。
#define va_arg(AP, TYPE)                                            \
 (AP += __va_rounded_size (TYPE),                                   \
  *((TYPE *) (AP - __va_rounded_size (TYPE))))

#endif /* _STDARG_H */

```

11.10 stddef.h 文件

11.10.1 功能描述

该文件定义了一些常用的类型和宏。内核中很少使用该文件。

11.10.2 代码注释

列表 linux/include/stddef.h 文件

```
#ifndef _STDDEF_H
#define _STDDEF_H

#ifndef _PTRDIFF_T
#define _PTRDIFF_T
typedef long ptrdiff_t;              // 两个指针相减结果的类型。
#endif

#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned long size_t;        // sizeof 返回的类型。
#endif

#undef NULL
#define NULL ((void *)0)             // 空指针。

#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)   // 成员在类型中的偏移位置。

#endif

```

11.11 string.h 文件

11.11.1 功能描述

该头文件中以内嵌函数的形式定义了所有字符串操作函数，为了提高执行速度使用了内嵌汇编程序。

11.11.2 代码注释

列表 linux/include/string.h 文件

```
#ifndef _STRING_H_
#define _STRING_H_

#ifndef NULL
#define NULL ((void *) 0)
#endif

#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;
#endif

extern char * strerror(int errno);

/*
 * This string-include defines all string functions as inline
 * functions. Use gcc. It also assumes ds=es=data space, this should be
 * normal. Most of the string-functions are rather heavily hand-optimized,
 * see especially strtok,strstr,str[c]spn. They should work, but are not
 * very easy to understand. Everything is done entirely within the register
 * set, making the functions fast and clean. String instructions have been
 * used through-out, making for "slightly" unclear code :-)
 *
 *              (C) 1991 Linus Torvalds
 */
/*
 * 这个字符串头文件以内嵌函数的形式定义了所有字符串操作函数。使用 gcc 时，同时
 * 假定了 ds=es=数据空间，这应该是常规的。绝大多数字符串函数都是经手工进行大量
 * 优化的，尤其是函数 strtok、strstr、str[c]spn。它们应该能正常工作，但却不是那
 * 么容易理解。所有的操作基本上都是使用寄存器集来完成的，这使得函数即快有整洁。
 * 所有地方都使用了字符串指令，这又使得代码“稍微”难以理解☺
 *
 *              (C) 1991 Linus Torvalds
 */

//// 将一个字符串(src)拷贝到另一个字符串(dest)，直到遇到 NULL 字符后停止。
// 参数：dest - 目的字符串指针，src - 源字符串指针。
// %0 - esi(src)，%1 - edi(dest)。
extern inline char * strcpy(char * dest,const char *src)
{
__asm__("cld\n"                     // 清方向位。
        "1:\tlodsb\n\t"             // 加载 DS:[esi]处 1 字节➔al，并更新 esi。
        "stosb\n\t"                 // 存储字节 al➔ES:[edi]，并更新 edi。
        "testb %%al,%%al\n\t"       // 刚存储的字节是 0?
        "jne 1b"                    // 不是则向后跳转到标号 1 处，否则结束。
        ::"S" (src),"D" (dest):"si","di","ax");
return dest;                        // 返回目的字符串指针。
}

//// 拷贝源字符串 count 个字节到目的字符串。
```

```
// 如果源串长度小于 count 个字节，就附加空字符(NULL)到目的字符串。
// 参数：dest - 目的字符串指针，src - 源字符串指针，count - 拷贝字节数。
// %0 - esi(src)，%1 - edi(dest)，%2 - ecx(count)。
extern inline char * strncpy(char * dest,const char *src,int count)
{
__asm__("cld\n"                     // 清方向位。
        "1:\tdecl %2\n\t"           // 寄存器 ecx--（count--）。
        "js 2f\n\t"                 // 如果 count<0 则向前跳转到标号 2，结束。
        "lodsb\n\t"                 // 取 ds:[esi]处 1 字节➔al，并且 esi++。
        "stosb\n\t"                 // 存储该字节➔es:[edi]，并且 edi++。
        "testb %%al,%%al\n\t"       // 该字节是 0?
        "jne 1b\n\t"                // 不是，则向前跳转到标号 1 处继续拷贝。
        "rep\n\t"                   // 否则，在目的串中存放剩余个数的空字符。
        "stosb\n"
        "2:"
        ::"S" (src),"D" (dest),"c" (count):"si","di","ax","cx");
return dest;                        // 返回目的字符串指针。
}

//// 将源字符串拷贝到目的字符串的末尾处。
// 参数：dest - 目的字符串指针，src - 源字符串指针。
// %0 - esi(src)，%1 - edi(dest)，%2 - eax(0)，%3 - ecx(-1)。
extern inline char * strcat(char * dest,const char * src)
{
__asm__("cld\n\t"                   // 清方向位。
        "repne\n\t"                 // 比较 al 与 es:[edi]字节，并更新 edi++，
        "scasb\n\t"                 // 直到找到目的串中是 0 的字节，此时 edi 已经指向后 1 字
节。
        "decl %1\n"                 // 让 es:[edi]指向 0 值字节。
        "1:\tlodsb\n\t"             // 取源字符串字节 ds:[esi]➔al，并 esi++。
        "stosb\n\t"                 // 将该字节存到 es:[edi]，并 edi++。
        "testb %%al,%%al\n\t"       // 该字节是 0?
        "jne 1b"                    // 不是，则向后跳转到标号 1 处继续拷贝，否则结束。
        ::"S" (src),"D" (dest),"a" (0),"c" (0xffffffff):"si","di","ax","cx");
return dest;                        // 返回目的字符串指针。
}

//// 将源字符串的 count 个字节复制到目的字符串的末尾处，最后添一空字符。
// 参数：dest - 目的字符串，src - 源字符串，count - 欲复制的字节数。
// %0 - esi(src)，%1 - edi(dest)，%2 - eax(0)，%3 - ecx(-1)，%4 - (count)。
extern inline char * strncat(char * dest,const char * src,int count)
{
__asm__("cld\n\t"                   // 清方向位。
        "repne\n\t"                 // 比较 al 与 es:[edi]字节，edi++。
        "scasb\n\t"                 // 直到找到目的串的末端 0 值字节。
        "decl %1\n\t"               // edi 指向该 0 值字节。
        "movl %4,%3\n"              // 欲复制字节数➔ecx。
        "1:\tdecl %3\n\t"           // ecx--（从 0 开始计数）。
        "js 2f\n\t"                 // ecx <0 ?，是则向前跳转到标号 2 处。
        "lodsb\n\t"                 // 否则取 ds:[esi]处的字节➔al，esi++。
        "stosb\n\t"                 // 存储到 es:[edi]处，edi++。
        "testb %%al,%%al\n\t"       // 该字节值为 0?
        "jne 1b\n"                  // 不是则向后跳转到标号 1 处，继续复制。
```

```
        "2:\txorl %2,%2\n\t"          // 将 al 清零。
        "stosb"                       // 存到 es:[edi]处。
        ::"S" (src),"D" (dest),"a" (0),"c" (0xffffffff),"g" (count)
        :"si","di","ax","cx");
return dest;                          // 返回目的字符串指针。
}

//// 将一个字符串与另一个字符串进行比较。
// 参数：cs - 字符串 1，ct - 字符串 2。
// %0 - eax(__res)返回值，%1 - edi(cs)字符串 1 指针，%2 - esi(ct)字符串 2 指针。
// 返回：如果串 1 > 串 2，则返回 1；串 1 = 串 2，则返回 0；串 1 < 串 2，则返回-1。
extern inline int strcmp(const char * cs,const char * ct)
{
register int __res __asm__("ax");     // __res 是寄存器变量(eax)。
__asm__("cld\n"                       // 清方向位。
        "1:\tlodsb\n\t"               // 取字符串 2 的字节 ds:[esi]➔al，并且 esi++。
        "scasb\n\t"                   // al 与字符串 1 的字节 es:[edi]作比较，并且 edi++。
        "jne 2f\n\t"                  // 如果不相等，则向前跳转到标号 2。
        "testb %%al,%%al\n\t"         // 该字节是 0 值字节吗（字符串结尾）？
        "jne 1b\n\t"                  // 不是，则向后跳转到标号 1，继续比较。
        "xorl %%eax,%%eax\n\t"        // 是，则返回值 eax 清零，
        "jmp 3f\n"                    // 向前跳转到标号 3，结束。
        "2:\tmovl $1,%%eax\n\t"       // eax 中置 1。
        "jl 3f\n\t"                   // 若前面比较中串 2 字符<串 1 字符，则返回正值，结束。
        "negl %%eax\n"                // 否则 eax = -eax，返回负值，结束。
        "3:"
        :"=a" (__res):"D" (cs),"S" (ct):"si","di");
return __res;                         // 返回比较结果。
}

//// 字符串 1 与字符串 2 的前 count 个字符进行比较。
// 参数：cs - 字符串 1，ct - 字符串 2，count - 比较的字符数。
// %0 - eax(__res)返回值，%1 - edi(cs)串 1 指针，%2 - esi(ct)串 2 指针，%3 - ecx(count)。
// 返回：如果串 1 > 串 2，则返回 1；串 1 = 串 2，则返回 0；串 1 < 串 2，则返回-1。
extern inline int strncmp(const char * cs,const char * ct,int count)
{
register int __res __asm__("ax");     // __res 是寄存器变量(eax)。
__asm__("cld\n"                       // 清方向位。
        "1:\tdecl %3\n\t"             // count--。
        "js 2f\n\t"                   // 如果 count<0，则向前跳转到标号 2。
        "lodsb\n\t"                   // 取串 2 的字符 ds:[esi]➔al，并且 esi++。
        "scasb\n\t"                   // 比较 al 与串 1 的字符 es:[edi]，并且 edi++。
        "jne 3f\n\t"                  // 如果不相等，则向前跳转到标号 3。
        "testb %%al,%%al\n\t"         // 该字符是 NULL 字符吗？
        "jne 1b\n"                    // 不是，则向后跳转到标号 1，继续比较。
        "2:\txorl %%eax,%%eax\n\t"    // 是 NULL 字符，则 eax 清零（返回值）。
        "jmp 4f\n"                    // 向前跳转到标号 4，结束。
        "3:\tmovl $1,%%eax\n\t"       // eax 中置 1。
        "jl 4f\n\t"                   // 如果前面比较中串 2 字符<串 2 字符，则返回 1，结束。
        "negl %%eax\n"                // 否则 eax = -eax，返回负值，结束。
        "4:"
        :"=a" (__res):"D" (cs),"S" (ct),"c" (count):"si","di","cx");
return __res;                         // 返回比较结果。
```

```
}

//// 在字符串中寻找第一个匹配的字符。
// 参数：s - 字符串，c - 欲寻找的字符。
// %0 - eax(__res)，%1 - esi(字符串指针 s)，%2 - eax(字符 c)。
// 返回：返回字符串中第一次出现匹配字符的指针。若没有找到匹配的字符，则返回空指针。
extern inline char * strchr(const char * s,char c)
{
register char * __res __asm__("ax"); // __res 是寄存器变量(eax)。
__asm__("cld\n\t"                    // 清方向位。
        "movb %%al,%%ah\n"           // 将欲比较字符移到 ah。
        "1:\tlodsb\n\t"              // 取字符串中字符 ds:[esi]➔al，并且 esi++。
        "cmpb %%ah,%%al\n\t"         // 字符串中字符 al 与指定字符 ah 相比较。
        "je 2f\n\t"                  // 若相等，则向前跳转到标号 2 处。
        "testb %%al,%%al\n\t"        // al 中字符是 NULL 字符吗？（字符串结尾？）
        "jne 1b\n\t"                 // 若不是，则向后跳转到标号 1，继续比较。
        "movl $1,%1\n"               // 是，则说明没有找到匹配字符，esi 置 1。
        "2:\tmovl %1,%0\n\t"         // 将指向匹配字符后一个字节处的指针值放入 eax
        "decl %0"                    // 将指针调整为指向匹配的字符。
        :"=a" (__res):"S" (s),"" (c):"si");
return __res;                        // 返回指针。
}

//// 寻找字符串中指定字符最后一次出现的地方。（反向搜索字符串）
// 参数：s - 字符串，c - 欲寻找的字符。
// %0 - edx(__res)，%1 - edx(0)，%2 - esi(字符串指针 s)，%3 - eax(字符 c)。
// 返回：返回字符串中最后一次出现匹配字符的指针。若没有找到匹配的字符，则返回空指针。
extern inline char * strrchr(const char * s,char c)
{
register char * __res __asm__("dx"); // __res 是寄存器变量(edx)。
__asm__("cld\n\t"                    // 清方向位。
        "movb %%al,%%ah\n"           // 将欲寻找的字符移到 ah。
        "1:\tlodsb\n\t"              // 取字符串中字符 ds:[esi]➔al，并且 esi++。
        "cmpb %%ah,%%al\n\t"         // 字符串中字符 al 与指定字符 ah 作比较。
        "jne 2f\n\t"                 // 若不相等，则向前跳转到标号 2 处。
        "movl %%esi,%0\n\t"          // 将字符指针保存到 edx 中。
        "decl %0\n"                  // 指针后退一位，指向字符串中匹配字符处。
        "2:\ttestb %%al,%%al\n\t"    // 比较的字符是 0 吗（到字符串尾）？
        "jne 1b"                     // 不是则向后跳转到标号 1 处，继续比较。
        :"=d" (__res):"" (0),"S" (s),"a" (c):"ax","si");
return __res;                        // 返回指针。
}

//// 在字符串 1 中寻找第 1 个字符序列，该字符序列中的任何字符都包含在字符串 2 中。
// 参数：cs - 字符串 1 指针，ct - 字符串 2 指针。
// %0 - esi(__res)，%1 - eax(0)，%2 - ecx(-1)，%3 - esi(串 1 指针 cs)，%4 - (串 2 指针 ct)。
// 返回字符串 1 中包含字符串 2 中任何字符的首个字符序列的长度值。
extern inline int strspn(const char * cs, const char * ct)
{
register char * __res __asm__("si"); // __res 是寄存器变量(esi)。
__asm__("cld\n\t"                    // 清方向位。
        "movl %4,%%edi\n\t"          // 首先计算串 2 的长度。串 2 指针放入 edi 中。
        "repne\n\t"                  // 比较 al(0)与串 2 中的字符（es:[edi]），并 edi++。
```

```
        "scasb\n\t"                  // 如果不相等就继续比较(ecx 逐步递减)。
        "notl %%ecx\n\t"             // ecx 中每位取反。
        "decl %%ecx\n\t"             // ecx--，得串 2 的长度值。
        "movl %%ecx,%%edx\n"         // 将串 2 的长度值暂放入 edx 中。
        "1:\tlodsb\n\t"              // 取串 1 字符 ds:[esi]➔al，并且 esi++。
        "testb %%al,%%al\n\t"        // 该字符等于 0 值吗（串 1 结尾）？
        "je 2f\n\t"                  // 如果是，则向前跳转到标号 2 处。
        "movl %4,%%edi\n\t"          // 取串 2 头指针放入 edi 中。
        "movl %%edx,%%ecx\n\t"       // 再将串 2 的长度值放入 ecx 中。
        "repne\n\t"                  // 比较 al 与串 2 中字符 es:[edi]，并且 edi++。
        "scasb\n\t"                  // 如果不相等就继续比较。
        "je 1b\n"                    // 如果相等，则向后跳转到标号 1 处。
        "2:\tdecl %0"                // esi--，指向最后一个包含在串 2 中的字符。
        :"=S" (__res):"a" (0),"c" (0xffffffff),"" (cs),"g" (ct)
        :"ax","cx","dx","di");
return __res-cs;                     // 返回字符序列的长度值。
}

//// 寻找字符串 1 中不包含字符串 2 中任何字符的首个字符序列。
// 参数：cs - 字符串 1 指针，ct - 字符串 2 指针。
// %0 - esi(__res)，%1 - eax(0)，%2 - ecx(-1)，%3 - esi(串 1 指针 cs)，%4 - (串 2 指针 ct)。
// 返回字符串 1 中不包含字符串 2 中任何字符的首个字符序列的长度值。
extern inline int strcspn(const char * cs, const char * ct)
{
register char * __res __asm__("si"); // __res 是寄存器变量(esi)。
__asm__("cld\n\t"                    // 清方向位。
        "movl %4,%%edi\n\t"          // 首先计算串 2 的长度。串 2 指针放入 edi 中。
        "repne\n\t"                  // 比较 al(0)与串 2 中的字符（es:[edi]），并 edi++。
        "scasb\n\t"                  // 如果不相等就继续比较(ecx 逐步递减)。
        "notl %%ecx\n\t"             // ecx 中每位取反。
        "decl %%ecx\n\t"             // ecx--，得串 2 的长度值。
        "movl %%ecx,%%edx\n"         // 将串 2 的长度值暂放入 edx 中。
        "1:\tlodsb\n\t"              // 取串 1 字符 ds:[esi]➔al，并且 esi++。
        "testb %%al,%%al\n\t"        // 该字符等于 0 值吗（串 1 结尾）？
        "je 2f\n\t"                  // 如果是，则向前跳转到标号 2 处。
        "movl %4,%%edi\n\t"          // 取串 2 头指针放入 edi 中。
        "movl %%edx,%%ecx\n\t"       // 再将串 2 的长度值放入 ecx 中。
        "repne\n\t"                  // 比较 al 与串 2 中字符 es:[edi]，并且 edi++。
        "scasb\n\t"                  // 如果不相等就继续比较。
        "jne 1b\n"                   // 如果不相等，则向后跳转到标号 1 处。
        "2:\tdecl %0"                // esi--，指向最后一个包含在串 2 中的字符。
        :"=S" (__res):"a" (0),"c" (0xffffffff),"" (cs),"g" (ct)
        :"ax","cx","dx","di");
return __res-cs;                     // 返回字符序列的长度值。
}

//// 在字符串 1 中寻找首个包含在字符串 2 中的任何字符。
// 参数：cs - 字符串 1 的指针，ct - 字符串 2 的指针。
// %0 -esi(__res)，%1 -eax(0)，%2 -ecx(0xffffffff)，%3 -esi(串 1 指针 cs)，%4 -(串 2 指针 ct)。
// 返回字符串 1 中首个包含字符串 2 中字符的指针。
extern inline char * strpbrk(const char * cs,const char * ct)
{
register char * __res __asm__("si"); // __res 是寄存器变量(esi)。
```

```
__asm__("cld\n\t"                     // 清方向位。
        "movl %4,%%edi\n\t"           // 首先计算串 2 的长度。串 2 指针放入 edi 中。
        "repne\n\t"                   // 比较 al(0)与串 2 中的字符（es:[edi]），并 edi++。
        "scasb\n\t"                   // 如果不相等就继续比较(ecx 逐步递减)。
        "notl %%ecx\n\t"              // ecx 中每位取反。
        "decl %%ecx\n\t"              // ecx--，得串 2 的长度值。
        "movl %%ecx,%%edx\n"          // 将串 2 的长度值暂放入 edx 中。
        "1:\tlodsb\n\t"               // 取串 1 字符 ds:[esi]➔al，并且 esi++。
        "testb %%al,%%al\n\t"         // 该字符等于 0 值吗（串 1 结尾）？
        "je 2f\n\t"                   // 如果是，则向前跳转到标号 2 处。
        "movl %4,%%edi\n\t"           // 取串 2 头指针放入 edi 中。
        "movl %%edx,%%ecx\n\t"        // 再将串 2 的长度值放入 ecx 中。
        "repne\n\t"                   // 比较 al 与串 2 中字符 es:[edi]，并且 edi++。
        "scasb\n\t"                   // 如果不相等就继续比较。
        "jne 1b\n\t"                  // 如果不相等，则向后跳转到标号 1 处。
        "decl %0\n\t"                 // esi--，指向一个包含在串 2 中的字符。
        "jmp 3f\n"                    // 向前跳转到标号 3 处。
        "2:\txorl %0,%0\n"            // 没有找到符合条件的，将返回值为 NULL。
        "3:"
        :"=S" (__res):"a" (0),"c" (0xffffffff),"" (cs),"g" (ct)
        :"ax","cx","dx","di");
return __res;                         // 返回指针值。
}

//// 在字符串 1 中寻找首个匹配整个字符串 2 的字符串。
// 参数：cs - 字符串 1 的指针，ct - 字符串 2 的指针。
// %0 -eax(__res)，%1 -eax(0)，%2 -ecx(0xffffffff)，%3 -esi(串 1 指针 cs)，%4 -(串 2 指针 ct)。
// 返回：返回字符串 1 中首个匹配字符串 2 的字符串指针。
extern inline char * strstr(const char * cs,const char * ct)
{
register char * __res __asm__("ax"); // __res 是寄存器变量(eax)。
__asm__("cld\n\t" \                   // 清方向位。
        "movl %4,%%edi\n\t"           // 首先计算串 2 的长度。串 2 指针放入 edi 中。
        "repne\n\t"                   // 比较 al(0)与串 2 中的字符（es:[edi]），并 edi++。
        "scasb\n\t"                   // 如果不相等就继续比较(ecx 逐步递减)。
        "notl %%ecx\n\t"              // ecx 中每位取反。
        "decl %%ecx\n\t"        /* NOTE! This also sets Z if searchstring='' */
                                /* 注意！如果搜索串为空，将设置 Z 标志 */ // 得串 2 的长度值。
        "movl %%ecx,%%edx\n"          // 将串 2 的长度值暂放入 edx 中。
        "1:\tmovl %4,%%edi\n\t"       // 取串 2 头指针放入 edi 中。
        "movl %%esi,%%eax\n\t"        // 将串 1 的指针复制到 eax 中。
        "movl %%edx,%%ecx\n\t"        // 再将串 2 的长度值放入 ecx 中。
        "repe\n\t"                    // 比较串 1 和串 2 字符(ds:[esi],es:[edi])，esi++,edi++。
        "cmpsb\n\t"                   // 若对应字符相等就一直比较下去。
        "je 2f\n\t"             /* also works for empty string, see above */
                                /* 对空串同样有效，见上面 */ // 若全相等，则转到标号 2。
        "xchgl %%eax,%%esi\n\t"       // 串 1 头指针➔esi，比较结果的串 1 指针➔eax。
        "incl %%esi\n\t"              // 串 1 头指针指向下一个字符。
        "cmpb $0,-1(%%eax)\n\t"       // 串 1 指针(eax-1)所指字节是 0 吗？
        "jne 1b\n\t"                  // 不是则跳转到标号 1，继续从串 1 的第 2 个字符开始比较。
        "xorl %%eax,%%eax\n\t"        // 清 eax，表示没有找到匹配。
        "2:"
        :"=a" (__res):"" (0),"c" (0xffffffff),"S" (cs),"g" (ct)
```

```
        :"cx","dx","di","si");
return __res;                       // 返回比较结果。
}

//// 计算字符串长度。
// 参数：s - 字符串。
// %0 - ecx(__res)，%1 - edi(字符串指针 s)，%2 - eax(0)，%3 - ecx(0xffffffff)。
// 返回：返回字符串的长度。
extern inline int strlen(const char * s)
{
register int __res __asm__("cx");   // __res 是寄存器变量(ecx)。
__asm__("cld\n\t"                   // 清方向位。
        "repne\n\t"                 // al(0)与字符串中字符 es:[edi]比较，
        "scasb\n\t"                 // 若不相等就一直比较。
        "notl %0\n\t"               // ecx 取反。
        "decl %0"                   // ecx--，得字符串得长度值。
        :"=c" (__res):"D" (s),"a" (0),"" (0xffffffff):"di");
return __res;                       // 返回字符串长度值。
}

extern char * ___strtok;       // 用于临时存放指向下面被分析字符串 1(s)的指针。

//// 利用字符串 2 中的字符将字符串 1 分割成标记(tokern)序列。
// 将串 1 看作是包含零个或多个单词(token)的序列，并由分割符字符串 2 中的一个或多个字符分开。
// 第一次调用 strtok()时，将返回指向字符串 1 中第 1 个 token 首字符的指针，并在返回 token 时将
// 一 null 字符写到分割符处。后续使用 null 作为字符串 1 的调用，将用这种方法继续扫描字符串 1，
// 直到没有 token 为止。在不同的调用过程中，分割符串 2 可以不同。
// 参数：s - 待处理的字符串 1，ct - 包含各个分割符的字符串 2。
// 汇编输出：%0 - ebx(__res)，%1 - esi(__strtok)；
// 汇编输入：%2 - ebx(__strtok)，%3 - esi(字符串 1 指针 s)，%4 - （字符串 2 指针 ct）。
// 返回：返回字符串 s 中第 1 个 token，如果没有找到 token，则返回一个 null 指针。
// 后续使用字符串 s 指针为 null 的调用，将在原字符串 s 中搜索下一个 token。
extern inline char * strtok(char * s,const char * ct)
{
register char * __res __asm__("si");
__asm__("testl %1,%1\n\t"           // 首先测试 esi(字符串 1 指针 s)是否是 NULL。
        "jne 1f\n\t"                // 如果不是，则表明是首次调用本函数，跳转标号 1。
        "testl %0,%0\n\t"          // 如果是 NULL，则表示此次是后续调用，测 ebx(__strtok)。
        "je 8f\n\t"                 // 如果 ebx 指针是 NULL，则不能处理，跳转结束。
        "movl %0,%1\n"              // 将 ebx 指针复制到 esi。
        "1:\txorl %0,%0\n\t"        // 清 ebx 指针。
        "movl $-1,%%ecx\n\t"        // 置 ecx = 0xffffffff。
        "xorl %%eax,%%eax\n\t"      // 清零 eax。
        "cld\n\t"                   // 清方向位。
        "movl %4,%%edi\n\t"         // 下面求字符串 2 的长度。edi 指向字符串 2。
        "repne\n\t"                 // 将 al(0)与 es:[edi]比较，并且 edi++。
        "scasb\n\t"                 // 直到找到字符串 2 的结束 null 字符，或计数 ecx==0。
        "notl %%ecx\n\t"            // 将 ecx 取反，
        "decl %%ecx\n\t"            // ecx--，得到字符串 2 的长度值。
        "je 7f\n\t"                     /* empty delimeter-string */
                                        /* 分割符字符串空 */ // 若串 2 长度为 0，则转标号 7。
        "movl %%ecx,%%edx\n"        // 将串 2 长度暂存入 edx。
        "2:\tlodsb\n\t"             // 取串 1 的字符 ds:[esi]➔al，并且 esi++。
```

```
        "testb %%al,%%al\n\t"        // 该字符为 0 值吗(串 1 结束)?
        "je 7f\n\t"                  // 如果是，则跳转标号 7。
        "movl %4,%%edi\n\t"          // edi 再次指向串 2 首。
        "movl %%edx,%%ecx\n\t"       // 取串 2 的长度值置入计数器 ecx。
        "repne\n\t"                  // 将 al 中串 1 的字符与串 2 中所有字符比较,
        "scasb\n\t"                  // 判断该字符是否为分割符。
        "je 2b\n\t"                  // 若能在串 2 中找到相同字符（分割符），则跳转标号 2。
        "decl %1\n\t"                // 若不是分割符，则串 1 指针 esi 指向此时的该字符。
        "cmpb $0,(%1)\n\t"           // 该字符是 NULL 字符吗?
        "je 7f\n\t"                  // 若是，则跳转标号 7 处。
        "movl %1,%0\n"               // 将该字符的指针 esi 存放在 ebx。
        "3:\tlodsb\n\t"              // 取串 1 下一个字符 ds:[esi]➔al，并且 esi++。
        "testb %%al,%%al\n\t"        // 该字符是 NULL 字符吗?
        "je 5f\n\t"                  // 若是，表示串 1 结束，跳转到标号 5。
        "movl %4,%%edi\n\t"          // edi 再次指向串 2 首。
        "movl %%edx,%%ecx\n\t"       // 串 2 长度值置入计数器 ecx。
        "repne\n\t"                  // 将 al 中串 1 的字符与串 2 中每个字符比较,
        "scasb\n\t"                  // 测试 al 字符是否是分割符。
        "jne 3b\n\t"                 // 若不是分割符则跳转标号 3，检测串 1 中下一个字符。
        "decl %1\n\t"                // 若是分割符，则 esi--，指向该分割符字符。
        "cmpb $0,(%1)\n\t"           // 该分割符是 NULL 字符吗?
        "je 5f\n\t"                  // 若是，则跳转到标号 5。
        "movb $0,(%1)\n\t"           // 若不是，则将该分割符用 NULL 字符替换掉。
        "incl %1\n\t"                // esi 指向串 1 中下一个字符，也即剩余串首。
        "jmp 6f\n"                   // 跳转标号 6 处。
        "5:\txorl %1,%1\n"           // esi 清零。
        "6:\tcmpb $0,(%0)\n\t"       // ebx 指针指向 NULL 字符吗?
        "jne 7f\n\t"                 // 若不是，则跳转标号 7。
        "xorl %0,%0\n"               // 若是，则让 ebx=NULL。
        "7:\ttestl %0,%0\n\t"        // ebx 指针为 NULL 吗?
        "jne 8f\n\t"                 // 若不是则跳转 8，结束汇编代码。
        "movl %0,%1\n"               // 将 esi 置为 NULL。
        "8:"
        :"=b" (__res),"=S" (___strtok)
        :"" (___strtok),"1" (s),"g" (ct)
        :"ax","cx","dx","di");
return __res;                        // 返回指向新 token 的指针。
}

//// 内存块复制。从源地址 src 处开始复制 n 个字节到目的地址 dest 处。
// 参数：dest - 复制的目的地址，src - 复制的源地址，n - 复制字节数。
// %0 - ecx(n)，%1 - esi(src)，%2 - edi(dest)。
extern inline void * memcpy(void * dest,const void * src, int n)
{
__asm__("cld\n\t"                    // 清方向位。
        "rep\n\t"                    // 重复执行复制 ecx 个字节,
        "movsb"                      // 从 ds:[esi]到 es:[edi]，esi++，edi++。
        ::"c" (n),"S" (src),"D" (dest)
        :"cx","si","di");
return dest;                         // 返回目的地址。
}

//// 内存块移动。同内存块复制，但考虑移动的方向。
```

```
// 参数：dest - 复制的目的地址，src - 复制的源地址，n - 复制字节数。
// 若 dest<src 则：%0 - ecx(n)，%1 - esi(src)，%2 - edi(dest)。
// 否则：%0 - ecx(n)，%1 - esi(src+n-1)，%2 - edi(dest+n-1)。
// 这样操作是为了防止在复制时错误地重叠覆盖。
extern inline void * memmove(void * dest,const void * src, int n)
{
if (dest<src)
__asm__("cld\n\t"                  // 清方向位。
        "rep\n\t"                  // 从 ds:[esi]到 es:[edi]，并且 esi++，edi++，
        "movsb"                    // 重复执行复制 ecx 字节。
        ::"c" (n),"S" (src),"D" (dest)
        :"cx","si","di");
else
__asm__("std\n\t"                  // 置方向位，从末端开始复制。
        "rep\n\t"                  // 从 ds:[esi]到 es:[edi]，并且 esi--，edi--，
        "movsb"                    // 复制 ecx 个字节。
        ::"c" (n),"S" (src+n-1),"D" (dest+n-1)
        :"cx","si","di");
return dest;
}

//// 比较 n 个字节的两块内存（两个字符串），即使遇上 NULL 字节也不停止比较。
// 参数：cs - 内存块 1 地址，ct - 内存块 2 地址，count - 比较的字节数。
// %0 - eax(__res)，%1 - eax(0)，%2 - edi(内存块 1)，%3 - esi(内存块 2)，%4 - ecx(count)。
// 返回：若块 1>块 2 返回 1；块 1<块 2，返回-1；块 1==块 2，则返回 0。
extern inline int memcmp(const void * cs,const void * ct,int count)
{
register int __res __asm__("ax");      // __res 是寄存器变量。
__asm__("cld\n\t"                  // 清方向位。
        "repe\n\t"                 // 如果相等则重复，
        "cmpsb\n\t"                // 比较 ds:[esi]与 es:[edi]的内容，并且 esi++，edi++。
        "je 1f\n\t"                // 如果都相同，则跳转到标号 1，返回 0(eax)值
        "movl $1,%%eax\n\t"        // 否则 eax 置 1，
        "jl 1f\n\t"                // 若内存块 2 内容的值<内存块 1，则跳转标号 1。
        "negl %%eax\n"             // 否则 eax = -eax。
        "1:"
        :"=a" (__res):"" (0),"D" (cs),"S" (ct),"c" (count)
        :"si","di","cx");
return __res;                      // 返回比较结果。
}

//// 在 n 字节大小的内存块(字符串)中寻找指定字符。
// 参数：cs - 指定内存块地址，c - 指定的字符，count - 内存块长度。
// %0 - edi(__res)，%1 - eax(字符 c)，%2 - edi(内存块地址 cs)，%3 - ecx(字节数 count)。
// 返回第一个匹配字符的指针，如果没有找到，则返回 NULL 字符。
extern inline void * memchr(const void * cs,char c,int count)
{
register void * __res __asm__("di"); // __res 是寄存器变量。
if (!count)                        // 如果内存块长度==0，则返回 NULL，没有找到。
        return NULL;
__asm__("cld\n\t"                  // 清方向位。
        "repne\n\t"                // 如果不相等则重复执行下面语句，
        "scasb\n\t"                // al 中字符与 es:[edi]字符作比较，并且 edi++，
```

```
        "je 1f\n\t"                 // 如果相等则向前跳转到标号 1 处。
        "movl $1,%0\n"              // 否则 edi 中置 1。
        "1:\tdecl %0"               // 让 edi 指向找到的字符（或是 NULL）。
        :"=D" (__res):"a" (c),"D" (cs),"c" (count)
        :"cx");
return __res;                       // 返回字符指针。
}

//// 用字符填写指定长度内存块。
// 用字符 c 填写 s 指向的内存区域，共填 count 字节。
// %0 - eax(字符 c)，%1 - edi(内存地址)，%2 - ecx(字节数 count)。
extern inline void * memset(void * s,char c,int count)
{
__asm__("cld\n\t"                   // 清方向位。
        "rep\n\t"                   // 重复 ecx 指定的次数，执行
        "stosb"                     // 将 al 中字符存入 es:[edi]中，并且 edi++。
        ::"a" (c),"D" (s),"c" (count)
        :"cx","di");
return s;
}

#endif
```

11.12 termios.h 文件

11.12.1 功能描述

该文件含有终端 I/O 接口定义。包括 termios 数据结构和一些对通用终端接口设置的函数原型。这些函数用来读取或设置终端的属性、线路控制、读取或设置波特率以及读取或设置终端前端进程的组 id。虽然这是 linux 早期的头文件，但已完全符合目前的 POSIX 标准，并作了适当的扩展。

在该文件中定义的两个终端数据结构 termio 和 termios 是分别属于两类 UNIX 系列(或刻隆)，termio 是在 AT&T 系统 V 中定义的，而 termios 是 POSIX 标准指定的。两个结构基本一样，只是 termio 使用短整数类型定义模式标志集，而 termios 使用长整数定义模式标志集。由于目前这两种结构都在使用，因此为了兼容性，大多数系统都同时支持它们。另外，以前使用的是一类似的 sgtty 结构，目前已基本不用。

11.12.2 代码注释

列表 linux/include/termios.h 文件

```
#ifndef _TERMIOS_H
#define _TERMIOS_H

#define TTY_BUF_SIZE 1024           // tty 中的缓冲区长度。

/* 0x54 is just a magic number to make these relatively uniqe ('T') */
```

```
/* 0x54 只是一个魔数，目的是为了使这些常数唯一('T') */

// tty 设备的 ioctl 调用命令集。ioctl 将命令编码在低位字中。
// 下面名称 TC[*]的含义是 tty 控制命令。
// 取相应终端 termios 结构中的信息(参见 tcgetattr())。
#define TCGETS          0x5401
// 设置相应终端 termios 结构中的信息(参见 tcsetattr()，TCSANOW)。
#define TCSETS          0x5402
// 在设置终端 termios 的信息之前，需要先等待输出队列中所有数据处理完(耗尽)。对于修改参数
// 会影响输出的情况，就需要使用这种形式(参见 tcsetattr()，TCSADRAIN 选项)。
#define TCSETSW         0x5403
// 在设置 termios 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新(清空)输入队列。
// 再设置（参见 tcsetattr()，TCSAFLUSH 选项）。
#define TCSETSF         0x5404
// 取相应终端 termio 结构中的信息(参见 tcgetattr())。
#define TCGETA          0x5405
// 设置相应终端 termio 结构中的信息(参见 tcsetattr()，TCSANOW 选项)。
#define TCSETA          0x5406
// 在设置终端 termio 的信息之前，需要先等待输出队列中所有数据处理完(耗尽)。对于修改参数
// 会影响输出的情况，就需要使用这种形式(参见 tcsetattr()，TCSADRAIN 选项)。
#define TCSETAW         0x5407
// 在设置 termio 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新(清空)输入队列。
// 再设置（参见 tcsetattr()，TCSAFLUSH 选项）。
#define TCSETAF         0x5408
// 等待输出队列处理完毕(空)，如果参数值是 0，则发送一个 break（参见 tcsendbreak()，tcdrain()）。
#define TCSBRK          0x5409
// 开始/停止控制。如果参数值是 0，则挂起输出；如果是 1，则重新开启挂起的输出；如果是 2，则挂
起
// 输入；如果是 3，则重新开启挂起的输入（参见 tcflow()）。
#define TCXONC          0x540A
//刷新已写输出但还没发送或已收但还没有读数据。如果参数是 0，则刷新(清空)输入队列；如果是 1，
// 则刷新输出队列；如果是 2，则刷新输入和输出队列（参见 tcflush()）。
#define TCFLSH          0x540B
// 下面名称 TIOC[*]的含义是 tty 输入输出控制命令。
// 设置终端串行线路专用模式。
#define TIOCEXCL        0x540C
// 复位终端串行线路专用模式。
#define TIOCNXCL        0x540D
// 设置 tty 为控制终端。(TIOCNOTTY - 禁止 tty 为控制终端)。
#define TIOCSCTTY       0x540E
// 读取指定终端设备进程的组 id(参见 tcgetpgrp())。
#define TIOCGPGRP       0x540F
// 设置指定终端设备进程的组 id(参见 tcsetpgrp())。
#define TIOCSPGRP       0x5410
// 返回输出队列中还未送出的字符数。
#define TIOCOUTQ        0x5411
// 模拟终端输入。该命令以一个指向字符的指针作为参数，并假装该字符是在终端上键入的。用户必须
// 在该控制终端上具有超级用户权限或具有读许可权限。
#define TIOCSTI         0x5412
// 读取终端设备窗口大小信息（参见 winsize 结构）。
#define TIOCGWINSZ      0x5413
// 设置终端设备窗口大小信息（参见 winsize 结构）。
#define TIOCSWINSZ      0x5414
```

```
    // 返回 modem 状态控制引线的当前状态比特位标志集（参见下面 185-196 行）。
28 #define TIOCMGET        0x5415
    // 设置单个 modem 状态控制引线的状态(true 或 false)(Individual control line Set)。
29 #define TIOCMBIS        0x5416
    // 复位单个 modem 状态控制引线的状态(Individual control line clear)。
30 #define TIOCMBIC        0x5417
    // 设置 modem 状态引线的状态。如果某一比特位置位，则 modem 对应的状态引线将置为有效。
31 #define TIOCMSET        0x5418
    // 读取软件载波检测标志(1 - 开启；0 - 关闭)。
    // 对于本地连接的终端或其它设备，软件载波标志是开启的，对于使用 modem 线路的终端或设备则
    // 是关闭的。为了能使用这两个 ioctl 调用，tty 线路应该是以 O_NDELAY 方式打开的，这样 open()
    // 就不会等待载波。
32 #define TIOCGSOFTCAR    0x5419
    // 设置软件载波检测标志(1 - 开启；0 - 关闭)。
33 #define TIOCSSOFTCAR    0x541A
    // 返回输入队列中还未取走字符的数目。
34 #define TIOCINQ         0x541B
35
    // 窗口大小(Window size)属性结构。在窗口环境中可用于基于屏幕的应用程序。
    // ioctls 中的 TIOCGWINSZ 和 TIOCSWINSZ 可用来读取或设置这些信息。
36 struct winsize {
37         unsigned short ws_row;       // 窗口字符行数。
38         unsigned short ws_col;       // 窗口字符列数。
39         unsigned short ws_xpixel;    // 窗口宽度，象素值。
40         unsigned short ws_ypixel;    // 窗口高度，象素值。
41 };
42
    // AT&T 系统 V 的 termio 结构。
43 #define NCC 8                        // termio 结构中控制字符数组的长度。
44 struct termio {
45         unsigned short c_iflag;         /* input mode flags */      // 输入模式标志。
46         unsigned short c_oflag;         /* output mode flags */     // 输出模式标志。
47         unsigned short c_cflag;         /* control mode flags */    // 控制模式标志。
48         unsigned short c_lflag;         /* local mode flags */      // 本地模式标志。
49         unsigned char c_line;           /* line discipline */       // 线路规程（速率）。
50         unsigned char c_cc[NCC];        /* control characters */    // 控制字符数组。
51 };
52
    // POSIX 的 termios 结构。
53 #define NCCS 17                      // termios 结构中控制字符数组的长度。
54 struct termios {
55         unsigned long c_iflag;          /* input mode flags */      // 输入模式标志。
56         unsigned long c_oflag;          /* output mode flags */     // 输出模式标志。
57         unsigned long c_cflag;          /* control mode flags */    // 控制模式标志。
58         unsigned long c_lflag;          /* local mode flags */      // 本地模式标志。
59         unsigned char c_line;           /* line discipline */       // 线路规程（速率）。
60         unsigned char c_cc[NCCS];       /* control characters */    // 控制字符数组。
61 };
62
63 /* c_cc characters */    /* c_cc 数组中的字符 */
    // 以下是 c_cc 数组对应字符的索引值。
64 #define VINTR 0          // c_cc[VINTR]    = INTR    (^C)，\003，中断字符。
65 #define VQUIT 1          // c_cc[VQUIT]    = QUIT    (^\)，\034，退出字符。
```

```
#define VERASE 2          // c_cc[VERASE]  = ERASE  (^H), \177, 擦出字符。
#define VKILL 3           // c_cc[VKILL]   = KILL   (^U), \025, 终止字符。
#define VEOF 4            // c_cc[VEOF]    = EOF    (^D), \004, 文件结束字符。
#define VTIME 5           // c_cc[VTIME]   = TIME   (\0), \0,   定时器值(参见后面说明)。
#define VMIN 6            // c_cc[VMIN]    = MIN    (\1), \1,   定时器值。
#define VSWTC 7           // c_cc[VSWTC]   = SWTC   (\0), \0,   交换字符。
#define VSTART 8          // c_cc[VSTART]  = START  (^Q), \021, 开始字符。
#define VSTOP 9           // c_cc[VSTOP]   = STOP   (^S), \023, 停止字符。
#define VSUSP 10          // c_cc[VSUSP]   = SUSP   (^Z), \032, 挂起字符。
#define VEOL 11           // c_cc[VEOL]    = EOL    (\0), \0,   行结束字符。
#define VREPRINT 12       // c_cc[VREPRINT] = REPRINT (^R), \022, 重显示字符。
#define VDISCARD 13       // c_cc[VDISCARD] = DISCARD (^O), \017, 丢弃字符。
#define VWERASE 14        // c_cc[VWERASE] = WERASE (^W), \027, 单词擦除字符。
#define VLNEXT 15         // c_cc[VLNEXT]  = LNEXT  (^V), \026, 下一行字符。
#define VEOL2 16          // c_cc[VEOL2]   = EOL2   (\0), \0,   行结束 2。

/* c_iflag bits */        /* c_iflag 比特位 */
// termios 结构输入模式字段 c_iflag 各种标志的符号常数。
#define IGNBRK  0000001             // 输入时忽略 BREAK 条件。
#define BRKINT  0000002             // 在 BREAK 时产生 SIGINT 信号。
#define IGNPAR  0000004             // 忽略奇偶校验出错的字符。
#define PARMRK  0000010             // 标记奇偶校验错。
#define INPCK   0000020             // 允许输入奇偶校验。
#define ISTRIP  0000040             // 屏蔽字符第 8 位。
#define INLCR   0000100             // 输入时将换行符 NL 映射成回车符 CR。
#define IGNCR   0000200             // 忽略回车符 CR。
#define ICRNL   0000400             // 在输入时将回车符 CR 映射成换行符 NL。
#define IUCLC   0001000             // 在输入时将大写字符转换成小写字符。
#define IXON    0002000             // 允许开始/停止（XON/XOFF）输出控制。
#define IXANY   0004000             // 允许任何字符重启输出。
#define IXOFF   0010000             // 允许开始/停止（XON/XOFF）输入控制。
#define IMAXBEL 0020000             // 输入队列满时响铃。

/* c_oflag bits */        /* c_oflag 比特位 */
// termios 结构中输出模式字段 c_oflag 各种标志的符号常数。
#define OPOST   0000001             // 执行输出处理。
#define OLCUC   0000002             // 在输出时将小写字符转换成大写字符。
#define ONLCR   0000004             // 在输出时将换行符 NL 映射成回车-换行符 CR-NL。
#define OCRNL   0000010             // 在输出时将回车符 CR 映射成换行符 NL。
#define ONOCR   0000020             // 在 0 列不输出回车符 CR。
#define ONLRET  0000040             // 换行符 NL 执行回车符的功能。
#define OFILL   0000100             // 延迟时使用填充字符而不使用时间延迟。
#define OFDEL   0000200             // 填充字符是 ASCII 码 DEL。如果未设置，则使用 ASCII NULL。
#define NLDLY   0000400             // 选择换行延迟。
#define   NL0   0000000             // 换行延迟类型 0。
#define   NL1   0000400             // 换行延迟类型 1。
#define CRDLY   0003000             // 选择回车延迟。
#define   CR0   0000000             // 回车延迟类型 0。
#define   CR1   0001000             // 回车延迟类型 1。
#define   CR2   0002000             // 回车延迟类型 2。
#define   CR3   0003000             // 回车延迟类型 3。
#define TABDLY  0014000             // 选择水平制表延迟。
#define   TAB0  0000000             // 水平制表延迟类型 0。
```

```
#define    TAB1   0004000           // 水平制表延迟类型 1。
#define    TAB2   0010000           // 水平制表延迟类型 2。
#define    TAB3   0014000           // 水平制表延迟类型 3。
#define    XTABS  0014000           // 将制表符 TAB 换成空格，该值表示空格数。
#define  BSDLY    0020000           // 选择退格延迟。
#define    BS0    0000000           // 退格延迟类型 0。
#define    BS1    0020000           // 退格延迟类型 1。
#define  VTDLY    0040000           // 纵向制表延迟。
#define    VT0    0000000           // 纵向制表延迟类型 0。
#define    VT1    0040000           // 纵向制表延迟类型 1。
#define  FFDLY    0040000           // 选择换页延迟。
#define    FF0    0000000           // 换页延迟类型 0。
#define    FF1    0040000           // 换页延迟类型 1。

/* c_cflag bit meaning */      /* c_cflag 比特位的含义 */
// termios 结构中控制模式标志字段 c_cflag 标志的符号常数（8 进制数）。
#define  CBAUD    0000017        // 传输速率位屏蔽码。
#define   B0      0000000        /* hang up */  /* 挂断线路 */
#define   B50     0000001        // 波特率 50。
#define   B75     0000002        // 波特率 75。
#define   B110    0000003        // 波特率 110。
#define   B134    0000004        // 波特率 134。
#define   B150    0000005        // 波特率 150。
#define   B200    0000006        // 波特率 200。
#define   B300    0000007        // 波特率 300。
#define   B600    0000010        // 波特率 600。
#define   B1200   0000011        // 波特率 1200。
#define   B1800   0000012        // 波特率 1800。
#define   B2400   0000013        // 波特率 2400。
#define   B4800   0000014        // 波特率 4800。
#define   B9600   0000015        // 波特率 9600。
#define   B19200  0000016        // 波特率 19200。
#define   B38400  0000017        // 波特率 38400。
#define  EXTA B19200             // 扩展波特率 A。
#define  EXTB B38400             // 扩展波特率 B。

#define  CSIZE    0000060        // 字符位宽度屏蔽码。
#define    CS5    0000000        // 每字符 5 比特位。
#define    CS6    0000020        // 每字符 6 比特位。
#define    CS7    0000040        // 每字符 7 比特位。
#define    CS8    0000060        // 每字符 8 比特位。
#define  CSTOPB   0000100        // 设置两个停止位，而不是 1 个。
#define  CREAD    0000200        // 允许接收。
#define  CPARENB  0000400        // 开启输出时产生奇偶位、输入时进行奇偶校验。
#define  CPARODD  0001000        // 输入/输入校验是奇校验。
#define  HUPCL    0002000        // 最后进程关闭后挂断。
#define  CLOCAL   0004000        // 忽略调制解调器(modem)控制线路。
#define  CIBAUD   03600000       /* input baud rate (not used) */  /* 输入波特率(未使用) */
#define  CRTSCTS  020000000000   /* flow control */  /* 流控制 */

#define  PARENB CPARENB          // 开启输出时产生奇偶位、输入时进行奇偶校验。
#define  PARODD CPARODD          // 输入/输入校验是奇校验。

```

```
/* c_lflag bits */        /* c_lflag 比特位 */
// termios 结构中本地模式标志字段 c_lflag 的符号常数。
#define ISIG    0000001          // 当收到字符 INTR、QUIT、SUSP 或 DSUSP，产生相应的信号。
#define ICANON  0000002          // 开启规范模式（熟模式）。
#define XCASE   0000004          // 若设置了 ICANON，则终端是大写字符的。
#define ECHO    0000010          // 回显输入字符。
#define ECHOE   0000020          // 若设置了 ICANON，则 ERASE/WERASE 将擦除前一字符/单词。
#define ECHOK   0000040          // 若设置了 ICANON，则 KILL 字符将擦除当前行。
#define ECHONL  0000100          // 如设置了 ICANON，则即使 ECHO 没有开启也回显 NL 字符。
#define NOFLSH  0000200          // 当生成 SIGINT 和 SIGQUIT 信号时不刷新输入输出队列，当
                                 // 生成 SIGSUSP 信号时，刷新输入队列。
#define TOSTOP  0000400          // 发送 SIGTTOU 信号到后台进程的进程组，该后台进程试图写
                                 // 自己的控制终端。
#define ECHOCTL 0001000          // 若设置了 ECHO，则除 TAB、NL、START 和 STOP 以外的 ASCII
                                 // 控制信号将被回显成象^X 式样，X 值是控制符+0x40。
#define ECHOPRT 0002000          // 若设置了 ICANON 和 IECHO，则字符在擦除时将显示。
#define ECHOKE  0004000          // 若设置了 ICANON，则 KILL 通过擦除行上的所有字符被回显。
#define FLUSHO  0010000          // 输出被刷新。通过键入 DISCARD 字符，该标志被翻转。
#define PENDIN  0040000          // 当下一个字符是读时，输入队列中的所有字符将被重显。
#define IEXTEN  0100000          // 开启实现时定义的输入处理。

/* modem lines */    /* modem 线路信号符号常数 */
#define TIOCM_LE        0x001        // 线路允许(Line Enable)。
#define TIOCM_DTR       0x002        // 数据终端就绪(Data Terminal Ready)。
#define TIOCM_RTS       0x004        // 请求发送(Request to Send)。
#define TIOCM_ST        0x008        // 串行数据发送(Serial Transfer)。[??]
#define TIOCM_SR        0x010        // 串行数据接收(Serial Receive)。[??]
#define TIOCM_CTS       0x020        // 清除发送(Clear To Send)。
#define TIOCM_CAR       0x040        // 载波监测(Carrier Detect)。
#define TIOCM_RNG       0x080        // 响铃指示(Ring indicate)。
#define TIOCM_DSR       0x100        // 数据设备就绪(Data Set Ready)。
#define TIOCM_CD        TIOCM_CAR
#define TIOCM_RI        TIOCM_RNG

/* tcflow() and TCXONC use these */    /* tcflow()和 TCXONC 使用这些符号常数 */
#define TCOOFF          0          // 挂起输出。
#define TCOON           1          // 重启被挂起的输出。
#define TCIOFF          2          // 系统传输一个 STOP 字符，使设备停止向系统传输数据。
#define TCION           3          // 系统传输一个 START 字符，使设备开始向系统传输数据。

/* tcflush() and TCFLSH use these */   /* tcflush()和 TCFLSH 使用这些符号常数 */
#define TCIFLUSH        0          // 清接收到的数据但不读。
#define TCOFLUSH        1          // 清已写的数据但不传送。
#define TCIOFLUSH       2          // 清接收到的数据但不读。清已写的数据但不传送。

/* tcsetattr uses these */             /* tcsetattr()使用这些符号常数 */
#define TCSANOW         0          // 改变立即发生。
#define TCSADRAIN       1          // 改变在所有已写的输出被传输之后发生。
#define TCSAFLUSH       2          // 改变在所有已写的输出被传输之后并且在所有接收到但
                                   // 还没有读取的数据被丢弃之后发生。

typedef int speed_t;               // 波特率数值类型。

```

```
    // 返回 termios_p 所指 termios 结构中的接收波特率。
extern speed_t cfgetispeed(struct termios *termios_p);
    // 返回 termios_p 所指 termios 结构中的发送波特率。
extern speed_t cfgetospeed(struct termios *termios_p);
    // 将 termios_p 所指 termios 结构中的接收波特率设置为 speed。
extern int cfsetispeed(struct termios *termios_p, speed_t speed);
    // 将 termios_p 所指 termios 结构中的发送波特率设置为 speed。
extern int cfsetospeed(struct termios *termios_p, speed_t speed);
    // 等待 fildes 所指对象已写输出数据被传送出去。
extern int tcdrain(int fildes);
    // 挂起/重启 fildes 所指对象数据的接收和发送。
extern int tcflow(int fildes, int action);
    // 丢弃 fildes 指定对象所有已写但还没传送以及所有已收到但还没有读取的数据。
extern int tcflush(int fildes, int queue_selector);
    // 获取与句柄 fildes 对应对象的参数，并将其保存在 termios_p 所指的地方。
extern int tcgetattr(int fildes, struct termios *termios_p);
    // 如果终端使用异步串行数据传输，则在一定时间内连续传输一系列 0 值比特位。
extern int tcsendbreak(int fildes, int duration);
    // 使用 termios 结构指针 termios_p 所指的数据，设置与终端相关的参数。
extern int tcsetattr(int fildes, int optional_actions,
        struct termios *termios_p);

#endif

```

11.12.3 其它信息

控制字符 TIME、MIN

在非规范模式输入处理中，输入字符没有被处理成行，因此擦除和终止处理也就不会发生。MIN 和 TIMEDE 的值即用于确定如何处理接收到的字符。

MIN 表示当满足读操作时（也即，当字符返给用户时）需要读取的最少字符数。TIME 是以 1/10 秒计数的定时值，用于超时定时和短期数据传输。这两个字符的四种组合情况及其相互作用描述如下：

MIN > 0，TIME > 0 的情况：

在这种情况下，TIME 起字符与字符间的定时器作用，并在接收到第 1 个字符后开始起作用。由于它是字符与字符间的定时器，所以在每收到一个字符就会被复位重启。MIN 与 TIME 之间的相互作用如下：一旦收到一个字符，字符间定时器就开始工作。如果在定时器超时（注意定时器每收到一个字符就会重新开始计时）之前收到了 MIN 个字符，则读操作即被满足。如果在 MIN 个字符被收到之前定时器超时了，就将到此时已收到的字符返回给用户。注意，如果 TIME 超时，则起码有一个接收到的字符将被返回，因为定时器只有在接收到了一个字符之后才开始起作用(计时)。在这种情况下(MIN > 0，TIME > 0)，读操作将会睡眠，直到接收到第 1 个字符激活 MIN 与 TIME 机制。如果读到字符数少于已有的字符数，那么定时器将不会被重新激活，因而随后的读操作将被立刻满足。

MIN > 0，TIME = 0 的情况：

在这种情况下，由于 TIME 的值是 0，因此定时器不起作用，只有 MIN 是有意义的。等待的读操作只有当接收到 MIN 个字符时才会被满足(等待着的操作将睡眠直到收到 MIN 个字符)。使用这种情况去读基于记录的终端 IO 的程序将会在读操作中被不确定地（随意地）阻塞。

MIN = 0，TIME > 0 的情况：

在这种情况下，由于 MIN=0，则 TIME 不再起字符间的定时器作用，而是一个读操作定时器，并在读操作一开始就起作用。只要接收到一个字符或者定时器超时就已满足读操作。注意，在这种情况下，如果定时器超时了，将读不到一个字符。如果定时器没有超时，那么只有在读到一个字符之后读操作才会满足。因此在这种情况下，读操作不会无限制地(不确定地)被阻塞，以等待字符。在读操作开始后，如果在 TIME*0.10 秒的时间内没有收到字符，读操作将以收到 0 个字符而返回。

MIN = 0，TIME = 0 的情况：

在这种情况下，读操作会立刻返回。所请求读的字符数或缓冲队列中现有字符数中的最小值将被返回，而不会等待更多的字符被输入缓冲中。

总得来说，在非规范模式下，这两个值是超时定时值和字符计数值。MIN 表示为了满足读操作，需要读取的最少字符数。TIME 是一个十分之一秒计数的计时值。当这两个都设置的话，读操作将等待，直到至少读到一个字符，然后在以读取 MIN 个字符或者时间 TIME 在读取最后一个字符后超时。如果仅设置了 MIN，那么在读取 MIN 个字符之前读操作将不返回。如果仅设置了 TIME，那么在读到至少一个字符或者定时超时后读操作将立刻返回。如果两个都没有设置，则读操作将立刻返回，仅给出目前已读的字节数。

11.13 time.h 文件

11.13.1 功能描述

该头文件用于涉及处理时间的函数。在 MINIX 中有一段对时间的描述很有趣：时间的处理较为复杂，比如什么是 GMT（格林威治标准时间）、本地时间或其它时间等。尽管主教 Ussher(1581-1656 年)曾经计算过，根据圣经，世界开始之日是公元前 4004 年 10 月 12 日上午 9 点，但在 UNIX 世界里，时间是从 GMT 1970 年 1 月 1 日午夜开始的，在这之前，所有均是空无的和(无效的)。

11.13.2 代码注释

列表 linux/include/time.h 文件

```
#ifndef _TIME_H
#define _TIME_H

#ifndef _TIME_T
#define _TIME_T
typedef long time_t;                    // 从 GMT 1970 年 1 月 1 日开始的以秒计数的时间(日历时间)。
#endif

#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;
#endif

#define CLOCKS_PER_SEC 100              // 系统时钟滴答频率，100HZ。
```

```
typedef long clock_t;                  // 从进程开始系统经过的时钟滴答数。

struct tm {
        int tm_sec;                    // 秒数 [0，59]。
        int tm_min;                    // 分钟数 [ 0，59]。
        int tm_hour;                   // 小时数 [0，59]。
        int tm_mday;                   // 1 个月的天数 [0，31]。
        int tm_mon;                    // 1 年中月份 [0，11]。
        int tm_year;                   // 从 1900 年开始的年数。
        int tm_wday;                   // 1 星期中的某天 [0，6]（星期天 =0）。
        int tm_yday;                   // 1 年中的某天 [0，365]。
        int tm_isdst;                  // 夏令时标志。
};

// 以下是有关时间操作的函数原型。
// 确定处理器使用时间。返回程序所用处理器时间（滴答数）的近似值。
clock_t clock(void);
// 取时间（秒数）。返回从 1970.1.1:0:0:0 开始的秒数（称为日历时间）。
time_t time(time_t * tp);
// 计算时间差。返回时间 time2 与 time1 之间经过的秒数。
double difftime(time_t time2, time_t time1);
// 将 tm 结构表示的时间转换成日历时间。
time_t mktime(struct tm * tp);

// 将 tm 结构表示的时间转换成一个字符串。返回指向该串的指针。
char * asctime(const struct tm * tp);
// 将日历时间转换成一个字符串形式，如“Wed Jun 30 21:49:08:1993\n”。
char * ctime(const time_t * tp);
// 将日历时间转换成 tm 结构表示的 UTC 时间（UTC - 世界时间代码 Universal Time Code）。
struct tm * gmtime(const time_t *tp);
// 将日历时间转换成 tm 结构表示的指定时间区(timezone)的时间。
struct tm *localtime(const time_t * tp);
// 将 tm 结构表示的时间利用格式字符串 fmt 转换成最大长度为 smax 的字符串并将结果存储在 s 中。
size_t strftime(char * s, size_t smax, const char * fmt, const struct tm * tp);
// 初始化时间转换信息，使用环境变量 TZ，对 zname 变量进行初始化。
// 在与时间区相关的时间转换函数中将自动调用该函数。
void tzset(void);

#endif
```

11.14 unistd.h 文件

11.14.1 功能描述

11.14.2 代码注释

列表 linux/include/unistd.h 文件

```
#ifndef _UNISTD_H
#define _UNISTD_H

/* ok, this may be a joke, but I'm working on it */
/* ok，这也许是个玩笑，但我正在着手处理 */
// 下面符号常数指出符合 IEEE 标准 1003.1 实现的版本号，是一个整数值。
#define _POSIX_VERSION 198808L

// chown()和 fchown()的使用受限于进程的权限。/* 只有超级用户可以执行 chown（我想..）*/
#define _POSIX_CHOWN_RESTRICTED /* only root can do a chown (I think..) */
// 长于(NAME_MAX)的路径名将产生错误，而不会自动截断。/* 路径名不截断（但是请看内核代码）*/
#define _POSIX_NO_TRUNC          /* no pathname truncation (but see in kernel) */
// 下面这个符号将定义成字符值，该值将禁止终端对其的处理。/* 禁止象^C 这样的字符 */
#define _POSIX_VDISABLE '\0'     /* character to disable things like ^C */
// 每个进程都有一保存的 set-user-ID 和一保存的 set-group-ID。 /* 我们将着手对此进行处理 */
/*#define _POSIX_SAVED_IDS */    /* we'll get to this yet */
// 系统实现支持作业控制。          /* 我们还没有支持这项标准，希望很快就行 */
/*#define _POSIX_JOB_CONTROL */ /* we aren't there quite yet. Soon hopefully */

#define STDIN_FILENO    0             // 标准输入文件句柄（描述符）号。
#define STDOUT_FILENO   1             // 标准输出文件句柄号。
#define STDERR_FILENO   2             // 标准出错文件句柄号。

#ifndef NULL
#define NULL    ((void *)0)           // 定义空指针。
#endif

/* access */    /* 文件访问 */
// 以下定义的符号常数用于 access()函数。
#define F_OK    0                     // 检测文件是否存在。
#define X_OK    1                     // 检测是否可执行（搜索）。
#define W_OK    2                     // 检测是否可写。
#define R_OK    4                     // 检测是否可读。

/* lseek */    /* 文件指针重定位 */
// 以下符号常数用于 lseek()和 fcntl()函数。
#define SEEK_SET        0             // 将文件读写指针设置为偏移值。
#define SEEK_CUR        1             // 将文件读写指针设置为当前值加上偏移值。
#define SEEK_END        2             // 将文件读写指针设置为文件长度加上偏移值。

/* _SC stands for System Configuration. We don't use them much */
/* _SC 表示系统配置。我们很少使用 */
// 下面的符号常数用于 sysconf()函数。
#define _SC_ARG_MAX             1     // 最大变量数。
#define _SC_CHILD_MAX           2     // 子进程最大数。
#define _SC_CLOCKS_PER_SEC      3     // 每秒滴答数。
#define _SC_NGROUPS_MAX         4     // 最大组数。
#define _SC_OPEN_MAX            5     // 最大打开文件数。
#define _SC_JOB_CONTROL         6     // 作业控制。
#define _SC_SAVED_IDS           7     // 保存的标识符。
```

```
#define _SC_VERSION             8     // 版本。

/* more (possibly) configurable things - now pathnames */
/* 更多的（可能的）可配置参数 - 现在用于路径名 */
// 下面的符号常数用于 pathconf()函数。
#define _PC_LINK_MAX            1     // 连接最大数。
#define _PC_MAX_CANON           2     // 最大常规文件数。
#define _PC_MAX_INPUT           3     // 最大输入长度。
#define _PC_NAME_MAX            4     // 名称最大长度。
#define _PC_PATH_MAX            5     // 路径最大长度。
#define _PC_PIPE_BUF            6     // 管道缓冲大小。
#define _PC_NO_TRUNC            7     // 文件名不截断。
#define _PC_VDISABLE            8     //
#define _PC_CHOWN_RESTRICTED    9     // 改变宿主受限。

#include <sys/stat.h>      // 文件状态头文件。含有文件或文件系统状态结构 stat{}和常量。
#include <sys/times.h>     // 定义了进程中运行时间结构 tms 以及 times()函数原型。
#include <sys/utsname.h>   // 系统名称结构头文件。
#include <utime.h>         // 用户时间头文件。定义了访问和修改时间结构以及 utime()原型。

#ifdef __LIBRARY__

// 以下是内核实现的系统调用符号常数，用于作为系统调用函数表中的索引值。
( include/linux/sys.h )
#define __NR_setup      0       /* used only by init, to get system going */
                                /* __NR_setup 仅用于初始化，以启动系统 */
#define __NR_exit       1
#define __NR_fork       2
#define __NR_read       3
#define __NR_write      4
#define __NR_open       5
#define __NR_close      6
#define __NR_waitpid    7
#define __NR_creat      8
#define __NR_link       9
#define __NR_unlink     10
#define __NR_execve     11
#define __NR_chdir      12
#define __NR_time       13
#define __NR_mknod      14
#define __NR_chmod      15
#define __NR_chown      16
#define __NR_break      17
#define __NR_stat       18
#define __NR_lseek      19
#define __NR_getpid     20
#define __NR_mount      21
#define __NR_umount     22
#define __NR_setuid     23
#define __NR_getuid     24
#define __NR_stime      25
#define __NR_ptrace     26
#define __NR_alarm      27
```

```
#define __NR_fstat      28
#define __NR_pause      29
#define __NR_utime      30
#define __NR_stty       31
#define __NR_gtty       32
#define __NR_access     33
#define __NR_nice       34
#define __NR_ftime      35
#define __NR_sync       36
#define __NR_kill       37
#define __NR_rename     38
#define __NR_mkdir      39
#define __NR_rmdir      40
#define __NR_dup        41
#define __NR_pipe       42
#define __NR_times      43
#define __NR_prof       44
#define __NR_brk        45
#define __NR_setgid     46
#define __NR_getgid     47
#define __NR_signal     48
#define __NR_geteuid    49
#define __NR_getegid    50
#define __NR_acct       51
#define __NR_phys       52
#define __NR_lock       53
#define __NR_ioctl      54
#define __NR_fcntl      55
#define __NR_mpx        56
#define __NR_setpgid    57
#define __NR_ulimit     58
#define __NR_uname      59
#define __NR_umask      60
#define __NR_chroot     61
#define __NR_ustat      62
#define __NR_dup2       63
#define __NR_getppid    64
#define __NR_getpgrp    65
#define __NR_setsid     66
#define __NR_sigaction  67
#define __NR_sgetmask   68
#define __NR_ssetmask   69
#define __NR_setreuid   70
#define __NR_setregid   71

    // 以下定义系统调用嵌入式汇编宏函数。
    // 不带参数的系统调用宏函数。type name(void)。
    // %0 - eax(__res)，%1 - eax(__NR_##name)。其中 name 是系统调用的名称，与 __NR_ 组合形成上面
    // 的系统调用符号常数，从而用来对系统调用表中函数指针寻址。
    // 返回：如果返回值大于等于 0，则返回该值，否则置出错号 errno，并返回-1。
#define _syscall0(type,name) \
type name(void) \
{ \
```

```
long __res; \
__asm__ volatile ("int $0x80" \                // 调用系统中断 0x80。
        : "=a" (__res) \                       // 返回值→eax(__res)。
        : "" (__NR_##name)); \                 // 输入为系统中断调用号__NR_name。
if (__res >= 0) \                              // 如果返回值>=0，则直接返回该值。
        return (type) __res; \
errno = -__res; \                              // 否则置出错号，并返回-1。
return -1; \
}

// 有 1 个参数的系统调用宏函数。type name(atype a)
// %0 - eax(__res)，%1 - eax(__NR_name)，%2 - ebx(a)。
#define _syscall1(type,name,atype,a) \
type name(atype a) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "" (__NR_##name),"b" ((long)(a))); \
if (__res >= 0) \
        return (type) __res; \
errno = -__res; \
return -1; \
}

// 有 2 个参数的系统调用宏函数。type name(atype a, btype b)
// %0 - eax(__res)，%1 - eax(__NR_name)，%2 - ebx(a)，%3 - ecx(b)。
#define _syscall2(type,name,atype,a,btype,b) \
type name(atype a,btype b) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "" (__NR_##name),"b" ((long)(a)),"c" ((long)(b))); \
if (__res >= 0) \
        return (type) __res; \
errno = -__res; \
return -1; \
}

// 有 3 个参数的系统调用宏函数。type name(atype a, btype b, ctype c)
// %0 - eax(__res)，%1 - eax(__NR_name)，%2 - ebx(a)，%3 - ecx(b)，%4 - edx(c)。
#define _syscall3(type,name,atype,a,btype,b,ctype,c) \
type name(atype a,btype b,ctype c) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "" (__NR_##name),"b" ((long)(a)),"c" ((long)(b)),"d" ((long)(c))); \
if (__res>=0) \
        return (type) __res; \
errno=-__res; \
return -1; \
```

```
}

#endif /* __LIBRARY__ */

extern int errno;                     // 出错号，全局变量。

// 对应各系统调用的函数原型定义。
int access(const char * filename, mode_t mode);
int acct(const char * filename);
int alarm(int sec);
int brk(void * end_data_segment);
void * sbrk(ptrdiff_t increment);
int chdir(const char * filename);
int chmod(const char * filename, mode_t mode);
int chown(const char * filename, uid_t owner, gid_t group);
int chroot(const char * filename);
int close(int fildes);
int creat(const char * filename, mode_t mode);
int dup(int fildes);
int execve(const char * filename, char ** argv, char ** envp);
int execv(const char * pathname, char ** argv);
int execvp(const char * file, char ** argv);
int execl(const char * pathname, char * arg0, ...);
int execlp(const char * file, char * arg0, ...);
int execle(const char * pathname, char * arg0, ...);
volatile void exit(int status);
volatile void _exit(int status);
int fcntl(int fildes, int cmd, ...);
int fork(void);
int getpid(void);
int getuid(void);
int geteuid(void);
int getgid(void);
int getegid(void);
int ioctl(int fildes, int cmd, ...);
int kill(pid_t pid, int signal);
int link(const char * filename1, const char * filename2);
int lseek(int fildes, off_t offset, int origin);
int mknod(const char * filename, mode_t mode, dev_t dev);
int mount(const char * specialfile, const char * dir, int rwflag);
int nice(int val);
int open(const char * filename, int flag, ...);
int pause(void);
int pipe(int * fildes);
int read(int fildes, char * buf, off_t count);
int setpgrp(void);
int setpgid(pid_t pid,pid_t pgid);
int setuid(uid_t uid);
int setgid(gid_t gid);
void (*signal(int sig, void (*fn)(int)))(int);
int stat(const char * filename, struct stat * stat_buf);
int fstat(int fildes, struct stat * stat_buf);
int stime(time_t * tptr);
```

```
int sync(void);
time_t time(time_t * tloc);
time_t times(struct tms * tbuf);
int ulimit(int cmd, long limit);
mode_t umask(mode_t mask);
int umount(const char * specialfile);
int uname(struct utsname * name);
int unlink(const char * filename);
int ustat(dev_t dev, struct ustat * ubuf);
int utime(const char * filename, struct utimbuf * times);
pid_t waitpid(pid_t pid,int * wait_stat,int options);
pid_t wait(int * wait_stat);
int write(int fildes, const char * buf, off_t count);
int dup2(int oldfd, int newfd);
int getppid(void);
pid_t getpgrp(void);
pid_t setsid(void);

#endif

```

11.15 utime.h 文件

11.15.1 功能描述

该文件定义了文件访问和修改时间结构 utimbuf{}以及 utime()函数原型。

11.15.2 代码注释

列表 linux/include/utime.h 文件

```
#ifndef _UTIME_H
#define _UTIME_H

#include <sys/types.h>  /* I know - shouldn't do this, but .. */
                        /* 我知道 - 不应该这样做，但是.. */
struct utimbuf {
        time_t actime;          // 文件访问时间。从 1970.1.1:0:0:0 开始的秒数。
        time_t modtime;         // 文件修改时间。从 1970.1.1:0:0:0 开始的秒数。
};

// 设置文件访问和修改时间函数。
extern int utime(const char *filename, struct utimbuf *times);

#endif

```

11.16 include/asm/目录下的文件

列表 linux/include/asm/目录下的文件

Name	Size	Last modified (GMT)	Description
io.h	477 bytes	1991-08-07 10:17:51	m
memory.h	507 bytes	1991-06-15 20:54:44	m
segment.h	1366 bytes	1991-11-25 18:48:24	m
system.h	1711 bytes	1991-09-17 13:08:31	m

11.17 io.h 文件

11.17.1 功能描述

该文件中定义了对硬件 IO 端口访问的嵌入式汇编宏函数：outb()、inb()以及 outb_p()和 inb_p()。前面两个函数与后面两个的主要区别在于后者代码中使用了 jmp 指令进行了时间延迟。

11.17.2 代码注释

列表 linux/include/asm/io.h 文件

```
//// 硬件端口字节输出函数。
// 参数：value - 欲输出字节；port - 端口。
#define outb(value,port) \
__asm__ ("outb %%al,%%dx"::"a" (value),"d" (port))


//// 硬件端口字节输入函数。
// 参数：port - 端口。返回读取的字节。
#define inb(port) ({ \
unsigned char _v; \
__asm__ volatile ("inb %%dx,%%al":"=a" (_v):"d" (port)); \
_v; \
})

//// 带延迟的硬件端口字节输出函数。
// 参数：value - 欲输出字节；port - 端口。
#define outb_p(value,port) \
__asm__ ("outb %%al,%%dx\n" \
```

```
                 "\tjmp 1f\n" \
                 "1:\tjmp 1f\n" \
                 "1:"::"a" (value),"d" (port))

//// 带延迟的硬件端口字节输入函数。
// 参数：port - 端口。返回读取的字节。
#define inb_p(port) ({ \
unsigned char _v; \
__asm__ volatile ("inb %%dx,%%al\n" \
        "\tjmp 1f\n" \
        "1:\tjmp 1f\n" \
        "1:":"=a" (_v):"d" (port)); \
_v; \
})

```

11.18 memory.h 文件

11.18.1 功能描述

该文件含有一个内存复制嵌入式汇编宏 memcpy()。与 string.h 中定义的 memcpy()相同，只是后者采用的是嵌入式汇编 C 函数形式定义的。

11.18.2 代码注释

列表 linux/include/asm/memory.h **文件**

```
/*
 * NOTE!!! memcpy(dest,src,n) assumes ds=es=normal data segment. This
 * goes for all kernel functions (ds=es=kernel space, fs=local data,
 * gs=null), as well as for all well-behaving user programs (ds=es=
 * user data space). This is NOT a bug, as any user program that changes
 * es deserves to die if it isn't careful.
 */
/*
 * 注意!!!memcpy(dest,src,n)假设段寄存器 ds=es=通常数据段。在内核中使用的
 * 所有函数都基于该假设（ds=es=内核空间，fs=局部数据空间，gs=null）,具有良好
 * 行为的应用程序也是这样（ds=es=用户数据空间）。如果任何用户程序随意改动了
 * es 寄存器而出错，则并不是由于系统程序错误造成的。
 */
//// 内存块复制。从源地址 src 处开始复制 n 个字节到目的地址 dest 处。
// 参数：dest - 复制的目的地址，src - 复制的源地址，n - 复制字节数。
// %0 - edi(目的地址 dest)，%1 - esi(源地址 src)，%2 - ecx(字节数 n)，
#define memcpy(dest,src,n) ({ \
void * _res = dest; \
__asm__ ("cld;rep;movsb" \                    // 从 ds:[esi]复制到 es:[edi]，并且 esi++，edi++。
                                              // 共复制 ecx(n)字节。
        ::"D" ((long)(_res)),"S" ((long)(src)),"c" ((long) (n)) \
        :"di","si","cx"); \
_res; \
```

```
})

```

11.19 segment.h 文件

11.19.1 功能描述

该文件中定义了一些访问段寄存器或与段寄存器有关的内存操作函数。

11.19.2 代码注释

列表 linux/include/asm/segment.h 文件

```
//// 读取 fs 段中指定地址处的字节。
// 参数：addr - 指定的内存地址。
// %0 - (返回的字节_v)；%1 - (内存地址 addr)。
// 返回：返回内存 fs:[addr]处的字节。
extern inline unsigned char get_fs_byte(const char * addr)
{
        unsigned register char _v;

        __asm__ ("movb %%fs:%1,%0":"=r" (_v):"m" (*addr));
        return _v;
}

//// 读取 fs 段中指定地址处的字。
// 参数：addr - 指定的内存地址。
// %0 - (返回的字_v)；%1 - (内存地址 addr)。
// 返回：返回内存 fs:[addr]处的字。
extern inline unsigned short get_fs_word(const unsigned short *addr)
{
        unsigned short _v;

        __asm__ ("movw %%fs:%1,%0":"=r" (_v):"m" (*addr));
        return _v;
}

//// 读取 fs 段中指定地址处的长字(4 字节)。
// 参数：addr - 指定的内存地址。
// %0 - (返回的长字_v)；%1 - (内存地址 addr)。
// 返回：返回内存 fs:[addr]处的长字。
extern inline unsigned long get_fs_long(const unsigned long *addr)
{
        unsigned long _v;

        __asm__ ("movl %%fs:%1,%0":"=r" (_v):"m" (*addr)); \
        return _v;
}

//// 将一字节存放在 fs 段中指定内存地址处。
```

```
// 参数：val - 字节值；addr - 内存地址。
// %0 - 寄存器(字节值 val)；%1 - (内存地址 addr)。
extern inline void put_fs_byte(char val,char *addr)
{
__asm__ ("movb %0,%%fs:%1"::"r" (val),"m" (*addr));
}

//// 将一字存放在 fs 段中指定内存地址处。
// 参数：val - 字值；addr - 内存地址。
// %0 - 寄存器(字值 val)；%1 - (内存地址 addr)。
extern inline void put_fs_word(short val,short * addr)
{
__asm__ ("movw %0,%%fs:%1"::"r" (val),"m" (*addr));
}

//// 将一长字存放在 fs 段中指定内存地址处。
// 参数：val - 长字值；addr - 内存地址。
// %0 - 寄存器(长字值 val)；%1 - (内存地址 addr)。
extern inline void put_fs_long(unsigned long val,unsigned long * addr)
{
__asm__ ("movl %0,%%fs:%1"::"r" (val),"m" (*addr));
}

/*
 * Someone who knows GNU asm better than I should double check the followig.
 * It seems to work, but I don't know if I'm doing something subtly wrong.
 * --- TYT, 11/24/91
 * [ nothing wrong here, Linus ]
 */
/*
 * 比我更懂 GNU 汇编的人应该仔细检查下面的代码。这些代码能使用，但我不知道是否
 * 含有一些小错误。
 * --- TYT，1991 年 11 月 24 日
 * [ 这些代码没有错误，Linus ]
 */

//// 取 fs 段寄存器值(选择符)。
// 返回：fs 段寄存器值。
extern inline unsigned long get_fs()
{
        unsigned short _v;
        __asm__("mov %%fs,%%ax":"=a" (_v):);
        return _v;
}

//// 取 ds 段寄存器值。
// 返回：ds 段寄存器值。
extern inline unsigned long get_ds()
{
        unsigned short _v;
        __asm__("mov %%ds,%%ax":"=a" (_v):);
        return _v;
}
```

```
//// 设置 fs 段寄存器。
// 参数：val - 段值（选择符）。
extern inline void set_fs(unsigned long val)
{
        __asm__("mov %0,%%fs"::"a" ((unsigned short) val));
}


```

11.20 system.h 文件

11.20.1 功能描述

该文件中定义了设置或修改描述符/中断门等的嵌入式汇编宏。

其中，函数 move_to_user_mode()是用于内核在初始化结束时切换到初始进程（任务 0）。所使用的方法是模拟中断调用返回过程，也即利用指令 iret 运行初始任务 0。

在切换到任务 0 之前，首先设置堆栈，模拟具有特权层切换的刚进入中断调用过程时堆栈的内容布置情况，见下图所示。然后执行 iret 指令，从而引起系统切换到任务 0 去执行。

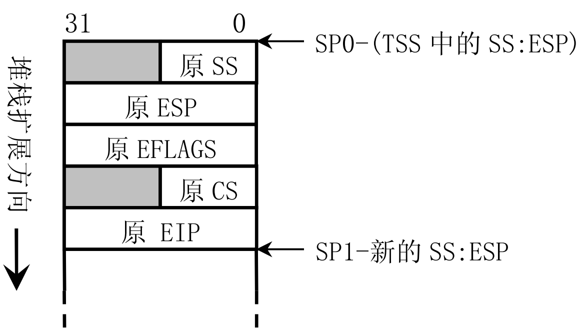


图 中断调用层间切换时堆栈内容

任务 0 是一个特殊进程，它的数据段和代码段直接映射到内核代码和数据空间，即从物理地址 0 开始的 640K 内存空间，其堆栈地址也即内核代码所使用的堆栈。因此图中堆栈中的原 SS 和原 ESP 是直接将现有内核的堆栈指针压入堆栈的。

11.20.2 代码注释

列表 linux/include/asm/system.h 文件

```
//// 切换到用户模式运行。
// 该函数利用 iret 指令实现从内核模式切换到用户模式（初始任务 0）。
#define move_to_user_mode() \
__asm__ ("movl %%esp,%%eax\n\t" \          // 保存堆栈指针 esp 到 eax 寄存器中。
        "pushl $0x17\n\t" \                // 首先将堆栈段选择符(SS)入栈。
```

```
        "pushl %%eax\n\t" \            // 然后将保存的堆栈指针值(esp)入栈。
        "pushfl\n\t" \                 // 将标志寄存器(eflags)内容入栈。
        "pushl $0x0f\n\t" \            // 将内核代码段选择符(cs)入栈。
        "pushl $1f\n\t" \              // 将下面标号 1 的偏移地址(eip)入栈。
        "iret\n" \                     // 执行中断返回指令，则会跳转到下面标号 1 处。
        "1:\tmovl $0x17,%%eax\n\t" \   // 此时开始执行任务 0，
        "movw %%ax,%%ds\n\t" \         // 初始化段寄存器指向本局部表的数据段。
        "movw %%ax,%%es\n\t" \
        "movw %%ax,%%fs\n\t" \
        "movw %%ax,%%gs" \
        :::"ax")

#define sti() __asm__ ("sti"::)        // 开中断嵌入汇编宏函数。
#define cli() __asm__ ("cli"::)        // 关中断。
#define nop() __asm__ ("nop"::)        // 空操作。

#define iret() __asm__ ("iret"::)      // 中断返回。

//// 设置门描述符宏函数。
// 参数：gate_addr -描述符地址；type -描述符中类型域值；dpl -描述符特权层值；addr -偏移地址。
// %0 - (由 dpl,type 组合成的类型标志字)；%1 - (描述符低 4 字节地址)；
// %2 - (描述符高 4 字节地址)；%3 - edx(程序偏移地址 addr)；%4 - eax(高字中含有段选择符)。
#define _set_gate(gate_addr,type,dpl,addr) \
__asm__ ("movw %%dx,%%ax\n\t" \        // 将偏移地址低字与选择符组合成描述符低 4 字节(eax)。
        "movw %0,%%dx\n\t" \           // 将类型标志字与偏移高字组合成描述符高 4 字节(edx)。
        "movl %%eax,%1\n\t" \          // 分别设置门描述符的低 4 字节和高 4 字节。
        "movl %%edx,%2" \
        : \
        : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
        "o" (*((char *) (gate_addr))), \
        "o" (*(4+(char *) (gate_addr))), \
        "d" ((char *) (addr)),"a" (0x00080000))

//// 设置中断门函数。
// 参数：n - 中断号；addr - 中断程序偏移地址。
// &idt[n]对应中断号在中断描述符表中的偏移值；中断描述符的类型是 14，特权级是 0。
#define set_intr_gate(n,addr) \
        _set_gate(&idt[n],14,0,addr)

//// 设置陷阱门函数。
// 参数：n - 中断号；addr - 中断程序偏移地址。
// &idt[n]对应中断号在中断描述符表中的偏移值；中断描述符的类型是 15，特权级是 0。
#define set_trap_gate(n,addr) \
        _set_gate(&idt[n],15,0,addr)

//// 设置系统调用门函数。
// 参数：n - 中断号；addr - 中断程序偏移地址。
// &idt[n]对应中断号在中断描述符表中的偏移值；中断描述符的类型是 15，特权级是 3。
#define set_system_gate(n,addr) \
        _set_gate(&idt[n],15,3,addr)

//// 设置段描述符函数。
// 参数：gate_addr -描述符地址；type -描述符中类型域值；dpl -描述符特权层值；
```

```
// base - 段的基地址；limit - 段限长。（参见段描述符的格式）
#define _set_seg_desc(gate_addr,type,dpl,base,limit) {\
        *(gate_addr) = ((base) & 0xff000000) | \             // 描述符低 4 字节。
                (((base) & 0x00ff0000)>>16) | \
                ((limit) & 0xf0000) | \
                ((dpl)<<13) | \
                (0x00408000) | \
                ((type)<<8); \
        *((gate_addr)+1) = (((base) & 0x0000ffff)<<16) | \   // 描述符高 4 字节。
                ((limit) & 0x0ffff); }

//// 在全局表中设置任务状态段/局部表描述符。
// 参数：n - 在全局表中描述符项 n 所对应的地址；addr - 状态段/局部表所在内存的基地址。
//       type - 描述符中的标志类型字节。
// %0 - eax(地址 addr)；%1 - (描述符项 n 的地址)；%2 - (描述符项 n 的地址偏移 2 处)；
// %3 - (描述符项 n 的地址偏移 4 处)；%4 - (描述符项 n 的地址偏移 5 处)；
// %5 - (描述符项 n 的地址偏移 6 处)；%6 - (描述符项 n 的地址偏移 7 处)；
#define _set_tssldt_desc(n,addr,type) \
__asm__ ("movw $104,%1\n\t" \          // 将 TSS 长度放入描述符长度域(第 0-1 字节)。
        "movw %%ax,%2\n\t" \           // 将基地址的低字放入描述符第 2-3 字节。
        "rorl $16,%%eax\n\t" \         // 将基地址高字移入 ax 中。
        "movb %%al,%3\n\t" \           // 将基地址高字中低字节移入描述符第 4 字节。
        "movb $" type ",%4\n\t" \      // 将标志类型字节移入描述符的第 5 字节。
        "movb $0x00,%5\n\t" \          // 描述符的第 6 字节置 0。
        "movb %%ah,%6\n\t" \           // 将基地址高字中高字节移入描述符第 7 字节。
        "rorl $16,%%eax" \             // eax 清零。
        ::"a" (addr), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)), \
         "m" (*(n+5)), "m" (*(n+6)), "m" (*(n+7)) \
        )

//// 在全局表中设置任务状态段描述符。
// n - 是该描述符的指针；addr - 是描述符中的基地址值。任务状态段描述符的类型是 0x89。
#define set_tss_desc(n,addr) _set_tssldt_desc(((char *) (n)),addr,"0x89")
//// 在全局表中设置局部表描述符。
// n - 是该描述符的指针；addr - 是描述符中的基地址值。局部表描述符的类型是 0x82。
#define set_ldt_desc(n,addr) _set_tssldt_desc(((char *) (n)),addr,"0x82")
```

11.21 include/linux/目录下的文件

列表 linux/include/linux/目录下的文件

Name	Size	Last modified (GMT)	Description
config.h	1289 bytes	1991-12-08 18:37:16	m
fdreg.h	2466 bytes	1991-11-02 10:48:44	m
fs.h	5474 bytes	1991-12-01 19:48:26	m
hdreg.h	1968 bytes	1991-10-13 15:32:15	m
head.h	304 bytes	1991-06-19 19:24:13	m
kernel.h	734 bytes	1991-12-02 03:19:07	m
mm.h	219 bytes	1991-07-29 17:51:12	m
sched.h	5838 bytes	1991-11-20 14:40:46	m
sys.h	2588 bytes	1991-11-25 20:15:35	m
tty.h	2173 bytes	1991-09-21 11:58:05	m

11.22 config.h 文件

11.22.1 功能描述

内核配置头文件。定义使用的键盘语言类型和硬盘类型（HD_TYPE）可选项。

11.22.2 代码注释

列表 linux/include/linux/config.h 文件

```
#ifndef _CONFIG_H
#define _CONFIG_H

/*
```

```
 * The root-device is no longer hard-coded. You can change the default
 * root-device by changing the line ROOT_DEV = XXX in boot/bootsect.s
 */
/*
 * 根文件系统设备已不再是硬编码的了。通过修改 boot/bootsect.s 文件中行
 * ROOT_DEV = XXX，你可以改变根设备的默认设置值。
 */

/*
 * define your keyboard here -
 * KBD_FINNISH for Finnish keyboards
 * KBD_US for US-type
 * KBD_GR for German keyboards
 * KBD_FR for Frech keyboard
 */
/*
 * 在这里定义你的键盘类型 -
 * KBD_FINNISH 是芬兰键盘。
 * KBD_US 是美式键盘。
 * KBD_GR 是德式键盘。
 * KBD_FR 是法式键盘。
 */
/*#define KBD_US */
/*#define KBD_GR */
/*#define KBD_FR */
#define KBD_FINNISH

/*
 * Normally, Linux can get the drive parameters from the BIOS at
 * startup, but if this for some unfathomable reason fails, you'd
 * be left stranded. For this case, you can define HD_TYPE, which
 * contains all necessary info on your harddisk.
 *
 * The HD_TYPE macro should look like this:
 *
 * #define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}
 *
 * In case of two harddisks, the info should be sepatated by
 * commas:
 *
 * #define HD_TYPE { h,s,c,wpcom,lz,ctl },{ h,s,c,wpcom,lz,ctl }
 */
/*
 * 通常，Linux 能够在启动时从 BIOS 中获取驱动器德参数，但是若由于未知原因
 * 而没有得到这些参数时，会使程序束手无策。对于这种情况，你可以定义 HD_TYPE，
 * 其中包括硬盘的所有信息。
 *
 * HD_TYPE 宏应该象下面这样的形式：
 *
 * #define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}
 *
 * 对于有两个硬盘的情况，参数信息需用逗号分开：
 *
```

```
 * #define HD_TYPE { h,s,c,wpcom,lz,ctl }, {h,s,c,wpcom,lz,ctl }
 */
/*
 This is an example, two drives, first is type 2, second is type 3:

#define HD_TYPE { 4,17,615,300,615,8 }, { 6,17,615,300,615,0 }

 NOTE: ctl is 0 for all drives with heads<=8, and ctl=8 for drives
 with more than 8 heads.

 If you want the BIOS to tell what kind of drive you have, just
 leave HD_TYPE undefined. This is the normal thing to do.
*/
/*
 * 下面是一个例子，两个硬盘，第 1 个是类型 2，第 2 个是类型 3：
 *
 * #define HD_TYPE { 4,17,615,300,615,8 }, {6,17,615,300,615,0 }
 *
 * 注意：对应所有硬盘，若其磁头数<=8，则 ctl 等于 0，若磁头数多于 8 个，
 * 则 ctl=8。
 *
 * 如果你想让 BIOS 给出硬盘的类型，那么只需不定义 HD_TYPE。这是默认操作。
 */

#endif

```

11.23 fdreg.h 头文件

11.23.1 功能描述

该头文件用以说明软盘系统常用到的一些参数以及所使用的 I/O 端口。由于软盘驱动器的控制比较烦琐，命令也多，因此在阅读代码之前，最好先参考有关微型计算机控制接口原理的书籍，了解软盘控制器(FDC)的工作原理，然后你就会觉得这里的定义还是比较合理有序的。

在编程时需要访问 4 个端口，分别对应一个或多个寄存器。对于 1.2M 的软盘控制器有以下一些端口。

表 软盘控制器端口

I/O 端口	读写性	寄存器名称
0x3f2	只写	数字输出寄存器(数字控制寄存器)
0x3f4	只读	FDC 主状态寄存器
0x3f5	读/写	FDC 数据寄存器
0x3f7	只读	数字输入寄存器
0x3f7	只写	磁盘控制寄存器(传输率控制)

数字输出端口（数字控制端口）是一个 8 位寄存器，它控制驱动器马达开启、驱动器选择、启动/复位 FDC 以及允许/禁止 DMA 及中断请求。

FDC 的主状态寄存器也是一个 8 位寄存器，用于反映软盘控制器 FDC 和软盘驱动器 FDD 的基本状态。通常，在 CPU 向 FDC 发送命令之前或从 FDC 获取操作结果之前，都要读取主状态寄存器的状态位，以判别当前 FDC 数据寄存器是否就绪，以及确定数据传送的方向。

FDC 的数据端口对应多个寄存器（只写型命令寄存器和参数寄存器、只读型结果寄存器），但任一时刻只能有一个寄存器出现在数据端口 0x3f5。在访问只写型寄存器时，主状态控制的 DIO 方向位必须为 0（CPU → FDC），访问只读型寄存器时则反之。在读取结果时只有在 FDC 不忙之后才算读完结果，通常结果数据最多有 7 个字节。

软盘控制器共可以接受 15 条命令。每个命令均经历三个阶段：命令阶段、执行阶段和结果阶段。

命令阶段是 CPU 向 FDC 发送命令字节和参数字节。每条命令的第一个字节总是命令字节（命令码）。其后跟着 0--8 字节的参数。执行阶段是 FDC 执行命令规定的操作。在执行阶段 CPU 是不加干预的，一般是通过 FDC 发出中断请求获知命令执行的结束。如果 CPU 发出的 FDC 命令是传送数据，则 FDC 可以以中断方式或 DMA 方式进行。中断方式每次传送 1 字节。DMA 方式是在 DMA 控制器管理下，FDC 与内存进行数据的传输直至全部数据传送完。此时 DMA 控制器会将传输字节计数终止信号通知 FDC，最后由 FDC 发出中断请求信号告知 CPU 执行阶段结束。结果阶段是由 CPU 读取 FDC 数据寄存器返回值，从而获得 FDC 命令执行的结果。返回结果数据的长度为 0--7 字节。对于没有返回结果数据的命令，则应向 FDC 发送检测中断状态命令获得操作的状态。

11.23.2 文件注释

列表 linux/include/linux/fdreg.h 文件

```
/*
 * This file contains some defines for the floppy disk controller.
 * Various sources. Mostly "IBM Microcomputers: A Programmers
 * Handbook", Sanches and Canton.
 */
/*
 * 该文件中含有一些软盘控制器的一些定义。这些信息有多处来源，大多数取自 Sanches 和 Canton
 * 编著的"IBM 微型计算机：程序员手册"一书。
 */
#ifndef _FDREG_H     // 该定义用来排除代码中重复包含此头文件。
#define _FDREG_H

// 一些软盘类型函数的原型说明。
extern int ticks_to_floppy_on(unsigned int nr);
extern void floppy_on(unsigned int nr);
extern void floppy_off(unsigned int nr);
extern void floppy_select(unsigned int nr);
extern void floppy_deselect(unsigned int nr);

// 下面是有关软盘控制器一些端口和符号的定义。
/* Fd controller regs. S&C, about page 340 */
/* 软盘控制器(FDC)积存器端口。摘自 S&C 书中约 340 页 */
#define FD_STATUS       0x3f4         // 主状态寄存器端口。
#define FD_DATA         0x3f5         // 数据端口。
#define FD_DOR          0x3f2         /* Digital Output Register */
                                      // 数字输出寄存器（也称为数字控制寄存器）。
#define FD_DIR          0x3f7         /* Digital Input Register (read) */
                                      // 数字输入寄存器。
#define FD_DCR          0x3f7         /* Diskette Control Register (write)*/
```

```
                                            // 数据传输率控制寄存器。

/* Bits of main status register */
/* 主状态寄存器各比特位的含义 */
#define STATUS_BUSYMASK 0x0F                /* drive busy mask */
                                            // 驱动器忙位（每位对应一个驱动器）。
#define STATUS_BUSY     0x10                /* FDC busy */
                                            // 软盘控制器忙。
#define STATUS_DMA      0x20                /* 0- DMA mode */
                                            // 0 – 为 DMA 数据传输模式，1 – 为非 DMA 模式。
#define STATUS_DIR      0x40                /* 0- cpu->fdc */
                                            // 传输方向：0 – CPU → fdc，1 – 相反。
#define STATUS_READY    0x80                /* Data reg ready */
                                            // 数据寄存器就绪位。

/* Bits of FD_ST0 */
/*状态字节 0（ST0）各比特位的含义 */
#define ST0_DS          0x03                /* drive select mask */
                                            // 驱动器选择号（发生中断时驱动器号）。
#define ST0_HA          0x04                /* Head (Address) */
                                            // 磁头号。
#define ST0_NR          0x08                /* Not Ready */
                                            // 磁盘驱动器未准备好。
#define ST0_ECE         0x10                /* Equipment chech error */
                                            // 设备检测出错（零磁道校准出错）。
#define ST0_SE          0x20                /* Seek end */
                                            // 寻道或重新校正操作执行结束。
#define ST0_INTR        0xC0                /* Interrupt code mask */
                                            // 中断代码位（中断原因），00 – 命令正常结束；
                                    // 01 – 命令异常结束；10 – 命令无效；11 – FDD 就绪状态改变。

/* Bits of FD_ST1 */
/*状态字节 1（ST1）各比特位的含义 */
#define ST1_MAM         0x01                /* Missing Address Mark */
                                            // 未找到地址标志(ID AM)。
#define ST1_WP          0x02                /* Write Protect */
                                            // 写保护。
#define ST1_ND          0x04                /* No Data - unreadable */
                                            // 未找到指定的扇区。
#define ST1_OR          0x10                /* OverRun */
                                            // 数据传输超时（DMA 控制器故障）。
#define ST1_CRC         0x20                /* CRC error in data or addr */
                                            // CRC 检验出错。
#define ST1_EOC         0x80                /* End Of Cylinder */
                                            // 访问超过一个磁道上的最大扇区号。

/* Bits of FD_ST2 */
/*状态字节 2（ST2）各比特位的含义 */
#define ST2_MAM         0x01                /* Missing Addess Mark (again) */
                                            // 未找到数据地址标志。
#define ST2_BC          0x02                /* Bad Cylinder */
                                            // 磁道坏。
#define ST2_SNS         0x04                /* Scan Not Satisfied */
```

```
                                        // 检索（扫描）条件不满足。
#define ST2_SEH         0x08            /* Scan Equal Hit */
                                        // 检索条件满足。
#define ST2_WC          0x10            /* Wrong Cylinder */
                                        // 磁道（柱面）号不符。
#define ST2_CRC         0x20            /* CRC error in data field */
                                        // 数据场 CRC 校验错。
#define ST2_CM          0x40            /* Control Mark = deleted */
                                        // 读数据遇到删除标志。

/* Bits of FD_ST3 */
/*状态字节 3（ST3）各比特位的含义 */
#define ST3_HA          0x04            /* Head (Address) */
                                        // 磁头号。
#define ST3_TZ          0x10            /* Track Zero signal (1=track 0) */
                                        // 零磁道信号。
#define ST3_WP          0x40            /* Write Protect */
                                        // 写保护。

/* Values for FD_COMMAND */
/* 软盘命令码 */
#define FD_RECALIBRATE  0x07            /* move to track 0 */
                                        // 重新校正(磁头退到零磁道)。
#define FD_SEEK         0x0F            /* seek track */
                                        // 磁头寻道。
#define FD_READ         0xE6            /* read with MT, MFM, SKip deleted */
                                        // 读数据（MT 多磁道操作，MFM 格式，跳过删除数据）。
#define FD_WRITE        0xC5            /* write with MT, MFM */
                                        // 写数据（MT，MFM）。
#define FD_SENSEI       0x08            /* Sense Interrupt Status */
                                        // 检测中断状态。
#define FD_SPECIFY      0x03            /* specify HUT etc */
                                        // 设定驱动器参数（步进速率、磁头卸载时间等）。

/* DMA commands */
/* DMA 命令 */
#define DMA_READ        0x46            // DMA 读盘，DMA 方式字（送 DMA 端口 12，11）。
#define DMA_WRITE       0x4A            // DMA 写盘，DMA 方式字。

#endif

```

11.24 fs.h 文件

11.24.1 功能描述

11.24.2 代码注释

列表 linux/include/linux/fs.h 文件

```
/*
 * This file has definitions for some important file table
 * structures etc.
 */
/*
 * 本文件含有某些重要文件表结构的定义等。
 */

#ifndef _FS_H
#define _FS_H

#include <sys/types.h>     // 类型头文件。定义了基本的系统数据类型。

/* devices are as follows: (same as minix, so we can use the minix
 * file system. These are major numbers.)
 *
 * 0 - unused (nodev)
 * 1 - /dev/mem
 * 2 - /dev/fd
 * 3 - /dev/hd
 * 4 - /dev/ttyx
 * 5 - /dev/tty
 * 6 - /dev/lp
 * 7 - unnamed pipes
 */
/*
 * 系统所含的设备如下：（与 minix 系统的一样，所以我们可以使用 minix 的
 * 文件系统。以下这些是主设备号。）
 *
 * 0 - 没有用到（nodev）
 * 1 - /dev/mem          内存设备。
 * 2 - /dev/fd           软盘设备。
 * 3 - /dev/hd           硬盘设备。
 * 4 - /dev/ttyx         tty 串行终端设备。
 * 5 - /dev/tty          tty 终端设备。
 * 6 - /dev/lp           打印设备。
 * 7 - unnamed pipes     没有命名的管道。
 */

#define IS_SEEKABLE(x) ((x)>=1 && (x)<=3)       // 是否是可以寻找定位的设备。

#define READ 0
#define WRITE 1
#define READA 2          /* read-ahead - don't pause */
#define WRITEA 3         /* "write-ahead" - silly, but somewhat useful */

void buffer_init(long buffer_end);

#define MAJOR(a) (((unsigned)(a))>>8)           // 取高字节（主设备号）。
#define MINOR(a) ((a)&0xff)                     // 取低字节（次设备号）。
```

```

#define NAME_LEN 14                         // 名字长度值。
#define ROOT_INO 1                          // 根 i 节点。

#define I_MAP_SLOTS 8                       // i 节点位图槽数。
#define Z_MAP_SLOTS 8                       // 逻辑块（区段块）位图槽数。
#define SUPER_MAGIC 0x137F                  // 文件系统魔数。

#define NR_OPEN 20                          // 打开文件数。
#define NR_INODE 32
#define NR_FILE 64
#define NR_SUPER 8
#define NR_HASH 307
#define NR_BUFFERS nr_buffers
#define BLOCK_SIZE 1024                     // 数据块长度。
#define BLOCK_SIZE_BITS 10                  // 数据块长度所占比特位数。
#ifndef NULL
#define NULL ((void *) 0)
#endif

// 每个逻辑块可存放的 i 节点数。
#define INODES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct d_inode)))
// 每个逻辑块可存放的目录项数。
#define DIR_ENTRIES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct dir_entry)))

// 管道头、管道尾、管道大小、管道空？、管道满？、管道头指针递增。
#define PIPE_HEAD(inode) ((inode).i_zone[0])
#define PIPE_TAIL(inode) ((inode).i_zone[1])
#define PIPE_SIZE(inode) ((PIPE_HEAD(inode)-PIPE_TAIL(inode))&(PAGE_SIZE-1))
#define PIPE_EMPTY(inode) (PIPE_HEAD(inode)==PIPE_TAIL(inode))
#define PIPE_FULL(inode) (PIPE_SIZE(inode)==(PAGE_SIZE-1))
#define INC_PIPE(head) \
__asm__("incl %0\n\tandl $4095,%0"::"m" (head))

typedef char buffer_block[BLOCK_SIZE];      // 块缓冲区。

// 缓冲区头数据结构。（极为重要！！！）
// 在程序中常用 bh 来表示 buffer_head 类型的缩写。
struct buffer_head {
        char * b_data;                  /* pointer to data block (1024 bytes) */  //数据块。
        unsigned long b_blocknr;        /* block number */      // 块号。
        unsigned short b_dev;           /* device (0 = free) */ // 数据源的设备号。
        unsigned char b_uptodate;       // 更新标志：表示数据是否已更新。
        unsigned char b_dirt;           /* 0-clean,1-dirty */ //修改标志：0-未修改，1-已修改。
        unsigned char b_count;          /* users using this block */   // 使用的用户数。
        unsigned char b_lock;           /* 0 - ok, 1 -locked */ // 缓冲区是否被锁定。
        struct task_struct * b_wait;    // 指向等待该缓冲区解锁的任务。
        struct buffer_head * b_prev;    // 前一块（这四个指针用于缓冲区的管理）。
        struct buffer_head * b_next;   // 下一块。
        struct buffer_head * b_prev_free;   // 前一空闲块。
        struct buffer_head * b_next_free;   // 下一空闲块。
};
```

```
// 磁盘上的索引节点(i 节点)数据结构。
struct d_inode {
        unsigned short i_mode;          // 文件类型和属性(rwx 位)。
        unsigned short i_uid;           // 用户 id（文件拥有者标识符）。
        unsigned long i_size;           // 文件大小（字节数）。
        unsigned long i_time;           // 修改时间（自 1970.1.1:0 算起，秒）。
        unsigned char i_gid;            // 组 id(文件拥有者所在的组)。
        unsigned char i_nlinks;         // 链接数（多少个文件目录项指向该 i 节点）。
        unsigned short i_zone[9];       // 直接(0-6)、间接(7)或双重间接(8)逻辑块号。
                                        // zone 是区的意思，可译成区段，或逻辑块。
};

// 这是在内存中的 i 节点结构。前 7 项与 d_inode 完全一样。
struct m_inode {
        unsigned short i_mode;          // 文件类型和属性(rwx 位)。
        unsigned short i_uid;           // 用户 id（文件拥有者标识符）。
        unsigned long i_size;           // 文件大小（字节数）。
        unsigned long i_mtime;          // 修改时间（自 1970.1.1:0 算起，秒）。
        unsigned char i_gid;            // 组 id(文件拥有者所在的组)。
        unsigned char i_nlinks;         // 文件目录项链接数。
        unsigned short i_zone[9];       // 直接(0-6)、间接(7)或双重间接(8)逻辑块号。
/* these are in memory also */
        struct task_struct * i_wait;    // 等待该 i 节点的进程。
        unsigned long i_atime;          // 最后访问时间。
        unsigned long i_ctime;          // i 节点自身修改时间。
        unsigned short i_dev;           // i 节点所在的设备号。
        unsigned short i_num;           // i 节点号。
        unsigned short i_count;         // i 节点被使用的次数，0 表示该 i 节点空闲。
        unsigned char i_lock;           // 锁定标志。
        unsigned char i_dirt;           // 已修改(脏)标志。
        unsigned char i_pipe;           // 管道标志。
        unsigned char i_mount;          // 安装标志。
        unsigned char i_seek;           // 搜寻标志(lseek 时)。
        unsigned char i_update;         // 更新标志。
};

// 文件结构（用于在文件句柄与 i 节点之间建立关系）
struct file {
        unsigned short f_mode;          // 文件操作模式（RW 位）
        unsigned short f_flags;         // 文件打开和控制的标志。
        unsigned short f_count;         // 对应文件句柄（文件描述符）数。
        struct m_inode * f_inode;       // 指向对应 i 节点。
        off_t f_pos;                    // 文件位置（读写偏移值）。
};

// 内存中磁盘超级块结构。
struct super_block {
        unsigned short s_ninodes;         // 节点数。
        unsigned short s_nzones;          // 逻辑块数。
        unsigned short s_imap_blocks;     // i 节点位图所占用的数据块数。
        unsigned short s_zmap_blocks;     // 逻辑块位图所占用的数据块数。
        unsigned short s_firstdatazone; // 第一个数据逻辑块号。
```

```
130         unsigned short s_log_zone_size; // log(数据块数/逻辑块)。（以 2 为底）。
131         unsigned long s_max_size;       // 文件最大长度。
132         unsigned short s_magic;         // 文件系统魔数。
133 /* These are only in memory */
134         struct buffer_head * s_imap[8]; // i 节点位图缓冲块指针数组(占用 8 块，可表示 64M)。
135         struct buffer_head * s_zmap[8]; // 逻辑块位图缓冲块指针数组（占用 8 块）。
136         unsigned short s_dev;           // 超级块所在的设备号。
137         struct m_inode * s_isup;        // 被安装的文件系统根目录的 i 节点。(isup-super i)
138         struct m_inode * s_imount;      // 被安装到的 i 节点。
139         unsigned long s_time;           // 修改时间。
140         struct task_struct * s_wait;    // 等待该超级块的进程。
141         unsigned char s_lock;           // 被锁定标志。
142         unsigned char s_rd_only;        // 只读标志。
143         unsigned char s_dirt;           // 已修改(脏)标志。
144 };
145
    // 磁盘上超级块结构。上面 125-132 行完全一样。
146 struct d_super_block {
147         unsigned short s_ninodes;       // 节点数。
148         unsigned short s_nzones;        // 逻辑块数。
149         unsigned short s_imap_blocks;   // i 节点位图所占用的数据块数。
150         unsigned short s_zmap_blocks;   // 逻辑块位图所占用的数据块数。
151         unsigned short s_firstdatazone; // 第一个数据逻辑块。
152         unsigned short s_log_zone_size; // log(数据块数/逻辑块)。（以 2 为底）。
153         unsigned long s_max_size;       // 文件最大长度。
154         unsigned short s_magic;         // 文件系统魔数。
155 };
156
    // 文件目录项结构。
157 struct dir_entry {
158         unsigned short inode;           // i 节点。
159         char name[NAME_LEN];            // 文件名。
160 };
161
162 extern struct m_inode inode_table[NR_INODE];     // 定义 i 节点表数组（32 项）。
163 extern struct file file_table[NR_FILE];          // 文件表数组（64 项）。
164 extern struct super_block super_block[NR_SUPER]; // 超级块数组（8 项）。
165 extern struct buffer_head * start_buffer;        // 缓冲区起始内存位置。
166 extern int nr_buffers;                           // 缓冲块数。
167
    //// 磁盘操作函数原型。
    // 检测驱动器中软盘是否改变。
168 extern void check_disk_change(int dev);
    // 检测指定软驱中软盘更换情况。如果软盘更换了则返回 1，否则返回 0。
169 extern int floppy_change(unsigned int nr);
    // 设置启动指定驱动器所需等待的时间（设置等待定时器）。
170 extern int ticks_to_floppy_on(unsigned int dev);
    // 启动指定驱动器。
171 extern void floppy_on(unsigned int dev);
    // 关闭指定的软盘驱动器。
172 extern void floppy_off(unsigned int dev);

    //// 以下是文件系统操作管理用的函数原型。
```

```
    // 将 i 节点指定的文件截为 0。
extern void truncate(struct m_inode * inode);
    // 刷新 i 节点信息。
extern void sync_inodes(void);
    // 等待指定的 i 节点。
extern void wait_on(struct m_inode * inode);
    // 逻辑块(区段，磁盘块)位图操作。取数据块 block 在设备上对应的逻辑块号。
extern int bmap(struct m_inode * inode,int block);
    // 创建数据块 block 在设备上对应的逻辑块，并返回在设备上的逻辑块号。
extern int create_block(struct m_inode * inode,int block);
    // 获取指定路径名的 i 节点号。
extern struct m_inode * namei(const char * pathname);
    // 根据路径名为打开文件操作作准备。
extern int open_namei(const char * pathname, int flag, int mode,
        struct m_inode ** res_inode);
    // 释放一个 i 节点(回写入设备)。
extern void iput(struct m_inode * inode);
    // 从设备读取指定节点号的一个 i 节点。
extern struct m_inode * iget(int dev,int nr);
    // 从 i 节点表(inode_table)中获取一个空闲 i 节点项。
extern struct m_inode * get_empty_inode(void);
    // 获取（申请一）管道节点。返回为 i 节点指针（如果是 NULL 则失败）。
extern struct m_inode * get_pipe_inode(void);
    // 在哈希表中查找指定的数据块。返回找到块的缓冲头指针。
extern struct buffer_head * get_hash_table(int dev, int block);
    // 从设备读取指定块（首先会在 hash 表中查找）。
extern struct buffer_head * getblk(int dev, int block);
    // 读/写数据块。
extern void ll_rw_block(int rw, struct buffer_head * bh);
    // 释放指定缓冲块。
extern void brelse(struct buffer_head * buf);
    // 读取指定的数据块。
extern struct buffer_head * bread(int dev,int block);
    // 读 4 块缓冲区到指定地址的内存中。
extern void bread_page(unsigned long addr,int dev,int b[4]);
    // 读取头一个指定的数据块，并标记后续将要读的块。
extern struct buffer_head * breada(int dev,int block,...);
    // 向设备 dev 申请一个磁盘块（区段，逻辑块）。返回逻辑块号
extern int new_block(int dev);
    // 释放设备数据区中的逻辑块(区段，磁盘块)block。复位指定逻辑块 block 的逻辑块位图比特位。
extern void free_block(int dev, int block);
    // 为设备 dev 建立一个新 i 节点，返回 i 节点号。
extern struct m_inode * new_inode(int dev);
    // 释放一个 i 节点（删除文件时）。
extern void free_inode(struct m_inode * inode);
    // 刷新指定设备缓冲区。
extern int sync_dev(int dev);
    // 读取指定设备的超级块。
extern struct super_block * get_super(int dev);
extern int ROOT_DEV;

    // 安装根文件系统。
extern void mount_root(void);
```

```

#endif

```

11.25 hdreg.h 文件

11.25.1 功能描述

11.25.2 代码注释

列表 linux/include/linux/hdreg.h 文件

```
/*
 * This file contains some defines for the AT-hd-controller.
 * Various sources. Check out some definitions (see comments with
 * a ques).
 */
/*
 * 本文件含有一些 AT 硬盘控制器的定义。来自各种资料。请查证某些
 * 定义（带有问号的注释）。
 */
#ifndef _HDREG_H
#define _HDREG_H

/* Hd controller regs. Ref: IBM AT Bios-listing */
/* 硬盘控制器寄存器端口。参见：IBM AT Bios 程序 */
#define HD_DATA         0x1f0   /* _CTL when writing */
#define HD_ERROR        0x1f1   /* see err-bits */
#define HD_NSECTOR      0x1f2   /* nr of sectors to read/write */
#define HD_SECTOR       0x1f3   /* starting sector */
#define HD_LCYL         0x1f4   /* starting cylinder */
#define HD_HCYL         0x1f5   /* high byte of starting cyl */
#define HD_CURRENT      0x1f6   /* 101dhhhh , d=drive, hhhh=head */
#define HD_STATUS       0x1f7   /* see status-bits */
#define HD_PRECOMP HD_ERROR     /* same io address, read=error, write=precomp */
#define HD_COMMAND HD_STATUS    /* same io address, read=status, write=cmd */

#define HD_CMD          0x3f6   // 控制寄存器端口。

/* Bits of HD_STATUS */
/* 硬盘状态寄存器各位的定义(HD_STATUS) */
#define ERR_STAT        0x01    // 命令执行错误。
#define INDEX_STAT      0x02    // 收到索引。
#define ECC_STAT        0x04    /* Corrected error */  // ECC 校验错。
#define DRQ_STAT        0x08    // 请求服务。
#define SEEK_STAT       0x10    // 寻道结束。
#define WRERR_STAT      0x20    // 驱动器故障。
#define READY_STAT      0x40    // 驱动器准备好（就绪）。
#define BUSY_STAT       0x80    // 控制器忙碌。
```

```

/* Values for HD_COMMAND */
/* 硬盘命令值（HD_CMD） */
#define WIN_RESTORE             0x10    // 驱动器重新校正（驱动器复位）。
#define WIN_READ                0x20    // 读扇区。
#define WIN_WRITE               0x30    // 写扇区。
#define WIN_VERIFY              0x40    // 扇区检验。
#define WIN_FORMAT              0x50    // 格式化磁道。
#define WIN_INIT                0x60    // 控制器初始化。
#define WIN_SEEK                0x70    // 寻道。
#define WIN_DIAGNOSE            0x90    // 控制器诊断。
#define WIN_SPECIFY             0x91    // 建立驱动器参数。

/* Bits for HD_ERROR */
/* 错误寄存器各比特位的含义（HD_ERROR） */
// 执行控制器诊断命令时含义与其它命令时的不同。下面分别列出：
// ==================================================
//          诊断命令时              其它命令时
// --------------------------------------------------
// 0x01     无错误                  数据标志丢失
// 0x02     控制器出错              磁道 0 错
// 0x03     扇区缓冲区错
// 0x04     ECC 部件错               命令放弃
// 0x05     控制处理器错
// 0x10                              ID 未找到
// 0x40                              ECC 错误
// 0x80                              坏扇区
//---------------------------------------------------
#define MARK_ERR        0x01    /* Bad address mark ? */
#define TRK0_ERR        0x02    /* couldn't find track 0 */
#define ABRT_ERR        0x04    /* ? */
#define ID_ERR          0x10    /* ? */
#define ECC_ERR         0x40    /* ? */
#define BBD_ERR         0x80    /* ? */

// 硬盘分区表结构。参见下面列表后信息。
struct partition {
        unsigned char boot_ind;         /* 0x80 - active (unused) */
        unsigned char head;             /* ? */
        unsigned char sector;           /* ? */
        unsigned char cyl;              /* ? */
        unsigned char sys_ind;          /* ? */
        unsigned char end_head;         /* ? */
        unsigned char end_sector;       /* ? */
        unsigned char end_cyl;          /* ? */
        unsigned int start_sect;        /* starting sector counting from 0 */
        unsigned int nr_sects;          /* nr of sectors in partition */
};

#endif

```

11.25.3 其它信息

硬盘分区表

为了实现多个操作系统共享硬盘资源，硬盘可以在逻辑上分为 1--4 个分区。每个分区之间的扇区号是邻接的。分区表由 4 个表项组成，每个表项由 16 字节组成，对应一个分区的信息，存放有分区的大小和起止的柱面号、磁道号和扇区号，见下表所示。分区表存放在硬盘的 0 柱面 0 头第 1 个扇区的 0x1BE--0x1FD 处。

表 硬盘分区表结构

位置	名称	大小	说明
0x00	boot_ind	字节	引导标志。4 个分区中同时只能有一个分区是可引导的。0x00-不从该分区引导操作系统；0x80-从该分区引导操作系统。
0x01	head	字节	分区起始磁头号。
0x02	sector	字节	分区起始扇区号(位 0-5)和起始柱面号高 2 位(位 6-7)。
0x03	cyl	字节	分区起始柱面号低 8 位。
0x04	sys_ind	字节	分区类型字节。0x0b-DOS；0x80-Old Minix；0x83-Linux …
0x05	end_head	字节	分区的结束磁头号。
0x06	end_sector	字节	结束扇区号(位 0-5)和结束柱面号高 2 位(位 6-7)。
0x07	end_cyl	字节	结束柱面号低 8 位。
0x08--0x0b	start_sect	长字	分区起始物理扇区号。
0x0c--0x0f	nr_sects	长字	分区占用的扇区数。

11.26 head.h 文件

11.26.1 功能描述

head 头文件，定义了 Intel CPU 中描述符的简单结构，和指定描述符的项号。

11.26.2 代码注释

列表 linux/include/linux/head.h 文件

```
#ifndef _HEAD_H
#define _HEAD_H

typedef struct desc_struct {         // 定义了段描述符的数据结构。该结构仅说明每个描述
        unsigned long a,b;           // 符是由 8 个字节构成，每个描述符表共有 256 项。
} desc_table[256];

extern unsigned long pg_dir[1024];  // 内存页目录数组。每个目录项为 4 字节。从物理地址 0 开始。
extern desc_table idt,gdt;           // 中断描述符表，全局描述符表。

#define GDT_NUL 0                    // 全局描述符表的第 0 项，不用。
```

```
#define GDT_CODE 1                    // 第 1 项，是内核代码段描述符项。
#define GDT_DATA 2                    // 第 2 项，是内核数据段描述符项。
#define GDT_TMP 3                     // 第 3 项，系统段描述符，Linux 没有使用。

#define LDT_NUL 0                     // 每个局部描述符表的第 0 项，不用。
#define LDT_CODE 1                    // 第 1 项，是用户程序代码段描述符项。
#define LDT_DATA 2                    // 第 2 项，是用户程序数据段描述符项。

#endif
```

11.27 kernel.h 文件

11.27.1 功能描述

定义了一些内核常用的函数原型等。

11.27.2 代码注释

列表 linux/include/linux/kernel.h

```
/*
 * 'kernel.h' contains some often-used function prototypes etc
 */
/*
 * 'kernel.h'定义了一些常用函数的原型等。
 */
// 验证给定地址开始的内存块是否超限。若超限则追加内存。( kernel/fork.c, 24 )。
void verify_area(void * addr,int count);
// 显示内核出错信息，然后进入死循环。( kernel/panic.c, 16 )。
volatile void panic(const char * str);
// 标准打印（显示）函数。( init/main.c, 151)。
int printf(const char * fmt, ...);
// 内核专用的打印信息函数，功能与 printf()相同。( kernel/printk.c, 21 )。
int printk(const char * fmt, ...);
// 往 tty 上写指定长度的字符串。( kernel/chr_drv/tty_io.c, 290 )。
int tty_write(unsigned ch,char * buf,int count);
// 通用内核内存分配函数。( lib/malloc.c, 117)。
void * malloc(unsigned int size);
// 释放指定对象占用的内存。( lib/malloc.c, 182)。
void free_s(void * obj, int size);

#define free(x) free_s((x), 0)

/*
 * This is defined as a macro, but at some point this might become a
 * real subroutine that sets a flag if it returns true (to do
 * BSD-style accounting where the process is flagged if it uses root
 * privs).  The implication of this is that you should do normal
 * permissions checks first, and check suser() last.
```

```
 */
/*
 * 下面函数是以宏的形式定义的，但是在某方面来看它可以成为一个真正的子程序，
 * 如果返回是 true 时它将设置标志（如果使用 root 用户权限的进程设置了标志，则用
 * 于执行 BSD 方式的计帐处理）。这意味着你应该首先执行常规权限检查，最后再
 * 检测 suser()。
 */
#define suser() (current->euid == 0)            // 检测是否是超级用户。


```

11.28 mm.h 文件

11.28.1 功能描述

mm.h 是内存管理头文件。其中主要定义了内存页面的大小和几个页面释放函数原型。

11.28.2 代码注释

列表 linux/include/linux/mm.h 文件

```
#ifndef _MM_H
#define _MM_H

#define PAGE_SIZE 4096          // 定义内存页面的大小(字节数)。

// 取空闲页面函数。返回页面地址。扫描页面映射数组 mem_map[]取空闲页面。
extern unsigned long get_free_page(void);
// 在指定物理地址处放置一页面。在页目录和页表中放置指定页面信息。
extern unsigned long put_page(unsigned long page,unsigned long address);
// 释放物理地址 addr 开始的一页面内存。修改页面映射数组 mem_map[]中引用次数信息。
extern void free_page(unsigned long addr);

#endif

```

11.29 sched.h 文件

11.29.1 功能描述

调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。

11.29.2 代码注释

列表 linux/include/linux/sched.h 文件

```
#ifndef _SCHED_H
```

```
#define _SCHED_H

#define NR_TASKS 64                   // 系统中同时最多任务（进程）数。
#define HZ 100                        // 定义系统时钟滴答频率(1 百赫兹，每个滴答 10ms)

#define FIRST_TASK task[0]            // 任务 0 比较特殊，所以特意给它单独定义一个符号。
#define LAST_TASK task[NR_TASKS-1]    // 任务数组中的最后一项任务。

#include <linux/head.h>    // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
#include <linux/fs.h>      // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）。
#include <linux/mm.h>      // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
#include <signal.h>        // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。

#if (NR_OPEN > 32)
#error "Currently the close-on-exec-flags are in one word, max 32 files/proc"
#endif

// 这里定义了进程运行可能处的状态。
#define TASK_RUNNING              0   // 进程正在运行或已准备就绪。
#define TASK_INTERRUPTIBLE        1   // 进程处于可中断等待状态。
#define TASK_UNINTERRUPTIBLE      2   // 进程处于不可中断等待状态，主要用于 I/O 操作等待。
#define TASK_ZOMBIE               3   // 进程处于僵死状态，已经停止运行，但父进程还没发信号。
#define TASK_STOPPED              4   // 进程已停止。

#ifndef NULL
#define NULL ((void *) 0)             // 定义 NULL 为空指针。
#endif

// 复制进程的页目录页表。Linus 认为这是内核中最复杂的函数之一。( mm/memory.c, 105 )
extern int copy_page_tables(unsigned long from, unsigned long to, long size);
// 释放页表所指定的内存块及页表本身。( mm/memory.c, 150 )
extern int free_page_tables(unsigned long from, unsigned long size);

// 调度程序的初始化函数。( kernel/sched.c, 385 )
extern void sched_init(void);
// 进程调度函数。( kernel/sched.c, 104 )
extern void schedule(void);
// 异常(陷阱)中断处理初始化函数，设置中断调用门并允许中断请求信号。( kernel/traps.c, 181 )
extern void trap_init(void);
// 显示内核出错信息，然后进入死循环。( kernel/panic.c, 16 )。
extern void panic(const char * str);
// 往 tty 上写指定长度的字符串。( kernel/chr_drv/tty_io.c, 290 )。
extern int tty_write(unsigned minor,char * buf,int count);

typedef int (*fn_ptr)();                     // 定义函数指针类型。

// 下面是数学协处理器使用的结构，主要用于保存进程切换时 i387 的执行状态信息。
struct i387_struct {
        long    cwd;        // 控制字(Control word)。
        long    swd;        // 状态字(Status word)。
        long    twd;        // 标记字(Tag word)。
        long    fip;        // 协处理器代码指针。
        long    fcs;        // 协处理器代码段寄存器。
```

```
        long    foo;
        long    fos;
        long    st_space[20];   /* 8*10 bytes for each FP-reg = 80 bytes */
};

// 任务状态段数据结构（参见列表后的信息）。
struct tss_struct {
        long    back_link;      /* 16 high bits zero */
        long    esp0;
        long    ss0;            /* 16 high bits zero */
        long    esp1;
        long    ss1;            /* 16 high bits zero */
        long    esp2;
        long    ss2;            /* 16 high bits zero */
        long    cr3;
        long    eip;
        long    eflags;
        long    eax,ecx,edx,ebx;
        long    esp;
        long    ebp;
        long    esi;
        long    edi;
        long    es;             /* 16 high bits zero */
        long    cs;             /* 16 high bits zero */
        long    ss;             /* 16 high bits zero */
        long    ds;             /* 16 high bits zero */
        long    fs;             /* 16 high bits zero */
        long    gs;             /* 16 high bits zero */
        long    ldt;            /* 16 high bits zero */
        long    trace_bitmap;   /* bits: trace 0, bitmap 16-31 */
        struct i387_struct i387;
};

// 这里是任务（进程）数据结构，或称为进程描述符。
// ==========================
// long state                   任务的运行状态（-1 不可运行，0 可运行(就绪)，>0 已停止）。
// long counter                 任务运行时间计数(递减)（滴答数），运行时间片。
// long priority                运行优先数。任务开始运行时 counter = priority，越大运行越长。
// long signal                  信号。是位图，每个比特位代表一种信号，信号值=位偏移值+1。
// struct sigaction sigaction[32] 信号执行属性结构，对应信号将要执行的操作和标志信息。
// long blocked                 进程信号屏蔽码（对应信号位图）。
// --------------------------
// int exit_code                任务执行停止的退出码，其父进程会取。
// unsigned long start_code     代码段地址。
// unsigned long end_code       代码长度（字节数）。
// unsigned long end_data       数据长度（字节数）。
// unsigned long brk            总长度（字节数）。
// unsigned long start_stack    堆栈段地址。
// long pid                     进程标识号(进程号)。
// long father                  父进程号。
// long pgrp                    父进程组号。
// long session                 会话号。
// long leader                  会话首领。
```

```
// unsigned short uid        用户标识号（用户 id）。
// unsigned short euid       有效用户 id。
// unsigned short suid       保存的用户 id。
// unsigned short gid        组标识号（组 id）。
// unsigned short egid       有效组 id。
// unsigned short sgid       保存的组 id。
// long alarm                报警定时值（滴答数）。
// long utime                用户态运行时间（滴答数）。
// long stime                系统态运行时间（滴答数）。
// long cutime               子进程用户态运行时间。
// long cstime               子进程系统态运行时间。
// long start_time           进程开始运行时刻。
// unsigned short used_math  标志：是否使用了协处理器。
// --------------------------
// int tty                   进程使用 tty 的子设备号。-1 表示没有使用。
// unsigned short umask      文件创建属性屏蔽位。
// struct m_inode * pwd      当前工作目录 i 节点结构。
// struct m_inode * root     根目录 i 节点结构。
// struct m_inode * executable 执行文件 i 节点结构。
// unsigned long close_on_exec 执行时关闭文件句柄位图标志。（参见 include/fcntl.h）
// struct file * filp[NR_OPEN] 进程使用的文件表结构。
// --------------------------
// struct desc_struct ldt[3] 本任务的局部表描述符。0-空，1-代码段 cs，2-数据和堆栈段 ds&ss。
// --------------------------
// struct tss_struct tss     本进程的任务状态段信息结构。
// ==========================
struct task_struct {
/* these are hardcoded - don't touch */
        long state;     /* -1 unrunnable, 0 runnable, >0 stopped */
        long counter;
        long priority;
        long signal;
        struct sigaction sigaction[32];
        long blocked;   /* bitmap of masked signals */
/* various fields */
        int exit_code;
        unsigned long start_code,end_code,end_data,brk,start_stack;
        long pid,father,pgrp,session,leader;
        unsigned short uid,euid,suid;
        unsigned short gid,egid,sgid;
        long alarm;
        long utime,stime,cutime,cstime,start_time;
        unsigned short used_math;
/* file system info */
        int tty;                /* -1 if no tty, so it must be signed */
        unsigned short umask;
        struct m_inode * pwd;
        struct m_inode * root;
        struct m_inode * executable;
        unsigned long close_on_exec;
        struct file * filp[NR_OPEN];
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
        struct desc_struct ldt[3];
```

```
/* tss for this task */
        struct tss_struct tss;
};

/*
 *  INIT_TASK is used to set up the first task table, touch at
 * your own risk!. Base=0, limit=0x9ffff (=640kB)
 */
/*
 * INIT_TASK 用于设置第 1 个任务表，若想修改，责任自负☺!
 * 基址 Base = 0，段长 limit = 0x9ffff（=640kB）。
 */
// 对应上面任务结构的第 1 个任务的信息。
#define INIT_TASK \
/* state etc */ { 0,15,15, \       // state, counter, priority
/* signals */   0,{{},},0, \       // signal, sigaction[32], blocked
/* ec,brk... */ 0,0,0,0,0,0, \     // exit_code,start_code,end_code,end_data,brk,start_stack
/* pid etc.. */ 0,-1,0,0,0, \      // pid, father, pgrp, session, leader
/* uid etc */   0,0,0,0,0,0, \     // uid, euid, suid, gid, egid, sgid
/* alarm */     0,0,0,0,0,0, \     // alarm, utime, stime, cutime, cstime, start_time
/* math */      0, \               // used_math
/* fs info */   -1,0022,NULL,NULL,NULL,0, \ // tty,umask,pwd,root,executable,close_on_exec
/* filp */      {NULL,}, \            // filp[20]
        { \                           // ldt[3]
                {0,0}, \
/* ldt */       {0x9f,0xc0fa00}, \  // 代码长 640K，基址 0x0，G=1，D=1，DPL=3，P=1 TYPE=0x0a
                {0x9f,0xc0f200}, \  // 数据长 640K，基址 0x0，G=1，D=1，DPL=3，P=1 TYPE=0x02
        }, \
/*tss*/ {0,PAGE_SIZE+(long)&init_task,0x10,0,0,0,0,(long)&pg_dir,\     // tss
         0,0,0,0,0,0,0,0, \
         0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
         _LDT(0),0x80000000, \
                {} \
        }, \
}

extern struct task_struct *task[NR_TASKS];       // 任务数组。
extern struct task_struct *last_task_used_math;  // 上一个使用过协处理器的进程。
extern struct task_struct *current;              // 当前进程结构指针变量。
extern long volatile jiffies;                    // 从开机开始算起的滴答数（10ms/滴答）。
extern long startup_time;                        // 开机时间。从 1970:0:0:0 开始计时的秒数。

#define CURRENT_TIME (startup_time+jiffies/HZ)   // 当前时间（秒数）。

// 添加定时器函数（定时时间 jiffies 滴答数，定时到时调用函数*fn()）。( kernel/sched.c,272)
extern void add_timer(long jiffies, void (*fn)(void));
// 不可中断的等待睡眠。( kernel/sched.c, 151 )
extern void sleep_on(struct task_struct ** p);
// 可中断的等待睡眠。( kernel/sched.c, 167 )
extern void interruptible_sleep_on(struct task_struct ** p);
// 明确唤醒睡眠的进程。( kernel/sched.c, 188 )
extern void wake_up(struct task_struct ** p);

```

```
/*
 * Entry into gdt where to find first TSS. 0-nul, 1-cs, 2-ds, 3-syscall
 * 4-TSS0, 5-LDT0, 6-TSS1 etc ...
 */
/*
 * 寻找第 1 个 TSS 在全局表中的入口。0-没有用 nul，1-代码段 cs，2-数据段 ds，3-系统段 syscall
 * 4-任务状态段 TSS0，5-局部表 LTD0，6-任务状态段 TSS1，等。
 */
// 全局表中第 1 个任务状态段(TSS)描述符的选择符索引号。
#define FIRST_TSS_ENTRY 4
// 全局表中第 1 个局部描述符表(LDT)描述符的选择符索引号。
#define FIRST_LDT_ENTRY (FIRST_TSS_ENTRY+1)
// 宏定义，计算在全局表中第 n 个任务的 TSS 描述符的索引号（选择符）。
#define _TSS(n) ((((unsigned long) n)<<4)+(FIRST_TSS_ENTRY<<3))
// 宏定义，计算在全局表中第 n 个任务的 LDT 描述符的索引号。
#define _LDT(n) ((((unsigned long) n)<<4)+(FIRST_LDT_ENTRY<<3))
// 宏定义，加载第 n 个任务的任务寄存器 tr。
#define ltr(n) __asm__("ltr %%ax"::"a" (_TSS(n)))
// 宏定义，加载第 n 个任务的局部描述符表寄存器 ldtr。
#define lldt(n) __asm__("lldt %%ax"::"a" (_LDT(n)))
// 取当前运行任务的任务号（是任务数组中的索引值，与进程号 pid 不同）。
// 返回：n - 当前任务号。用于( kernel/traps.c, 79)。
#define str(n) \
__asm__("str %%ax\n\t" \              // 将任务寄存器中 TSS 段的有效地址➔ax
        "subl %2,%%eax\n\t" \         // (eax - FIRST_TSS_ENTRY*8)➔eax
        "shrl $4,%%eax" \             // (eax/16)➔eax = 当前任务号。
        :"=a" (n) \
        :"a" (0),"i" (FIRST_TSS_ENTRY<<3))
/*
 *      switch_to(n) should switch tasks to task nr n, first
 * checking that n isn't the current task, in which case it does nothing.
 * This also clears the TS-flag if the task we switched to has used
 * tha math co-processor latest.
 */
/*
 * switch_to(n)将切换当前任务到任务 nr，即 n。首先检测任务 n 不是当前任务，
 * 如果是则什么也不做退出。如果我们切换到的任务最近（上次运行）使用过数学
 * 协处理器的话，则还需复位控制寄存器 cr0 中的 TS 标志。
 */
// 输入：%0 - 新 TSS 的偏移地址(*&__tmp.a)； %1 - 存放新 TSS 的选择符值(*&__tmp.b)；
//       dx - 新任务 n 的选择符；ecx - 新任务指针 task[n]。
// 其中临时数据结构__tmp 中，a 的值是 32 位偏移值，b 为新 TSS 的选择符。在任务切换时，a 值
// 没有用（忽略）。在判断新任务上次执行是否使用过协处理器时，是通过将新任务状态段的地址与
// 保存在 last_task_used_math 变量中的使用过协处理器的任务状态段的地址进行比较而作出的。
#define switch_to(n) {\
struct {long a,b;} __tmp; \
__asm__("cmpl %%ecx,_current\n\t" \      // 任务 n 是当前任务吗?(current ==task[n]?)
        "je 1f\n\t" \                    // 是，则什么都不做，退出。
        "movw %%dx,%1\n\t" \             // 将新任务的选择符➔*&__tmp.b。
        "xchgl %%ecx,_current\n\t" \     // current = task[n]；ecx = 被切换出的任务。
        "ljmp %0\n\t" \                  // 执行长跳转至*&__tmp，造成任务切换。
                                         // 在任务切换回来后才会继续执行下面的语句。
        "cmpl %%ecx,_last_task_used_math\n\t" \ // 新任务上次使用过协处理器吗？
```

```
        "jne 1f\n\t" \                  // 没有则跳转，退出。
        "clts\n" \                      // 新任务上次使用过协处理器，则清 cr0 的 TS 标志。
        "1:" \
        ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
        "d" (_TSS(n)),"c" ((long) task[n])); \
}

// 页面地址对准。（在内核代码中没有任何地方引用!!）
#define PAGE_ALIGN(n) (((n)+0xfff)&0xfffff000)

// 设置位于地址 addr 处描述符中的各基地址字段(基地址是 base)，参见列表后说明。
// %0 - 地址 addr 偏移 2；%1 - 地址 addr 偏移 4；%2 - 地址 addr 偏移 7；edx - 基地址 base。
#define _set_base(addr,base) \
__asm__("movw %%dx,%0\n\t" \           // 基址 base 低 16 位(位 15-0)➔[addr+2]。
        "rorl $16,%%edx\n\t" \          // edx 中基址高 16 位(位 31-16)➔dx。
        "movb %%dl,%1\n\t" \            // 基址高 16 位中的低 8 位(位 23-16)➔[addr+4]。
        "movb %%dh,%2" \                // 基址高 16 位中的高 8 位(位 31-24)➔[addr+7]。
        ::"m" (*((addr)+2)), \
          "m" (*((addr)+4)), \
          "m" (*((addr)+7)), \
          "d" (base) \
        :"dx")

// 设置位于地址 addr 处描述符中的段限长字段(段长是 limit)。
// %0 - 地址 addr；%1 - 地址 addr 偏移 6 处；edx - 段长值 limit。
#define _set_limit(addr,limit) \
__asm__("movw %%dx,%0\n\t" \           // 段长 limit 低 16 位(位 15-0)➔[addr]。
        "rorl $16,%%edx\n\t" \          // edx 中的段长高 4 位(位 19-16)➔dl。
        "movb %1,%%dh\n\t" \            // 取原[addr+6]字节➔dh，其中高 4 位是些标志。
        "andb $0xf0,%%dh\n\t" \         // 清 dh 的低 4 位(将存放段长的位 19-16)。
        "orb %%dh,%%dl\n\t" \           // 将原高 4 位标志和段长的高 4 位(位 19-16)合成 1 字节，
        "movb %%dl,%1" \                // 并放会[addr+6]处。
        ::"m" (*(addr)), \
          "m" (*((addr)+6)), \
          "d" (limit) \
        :"dx")

// 设置局部描述符表中 ldt 描述符的基地址字段。
#define set_base(ldt,base) _set_base( ((char *)&(ldt)) , base )
// 设置局部描述符表中 ldt 描述符的段长字段。
#define set_limit(ldt,limit) _set_limit( ((char *)&(ldt)) , (limit-1)>>12 )

// 从地址 addr 处描述符中取段基地址。功能与_set_base()正好相反。
// edx - 存放基地址(__base)；%1 - 地址 addr 偏移 2；%2 - 地址 addr 偏移 4；%3 - addr 偏移 7。
#define _get_base(addr) ({\
unsigned long __base; \
__asm__("movb %3,%%dh\n\t" \           // 取[addr+7]处基地址高 16 位的高 8 位(位 31-24)➔dh。
        "movb %2,%%dl\n\t" \            // 取[addr+4]处基址高 16 位的低 8 位(位 23-16)➔dl。
        "shll $16,%%edx\n\t" \          // 基地址高 16 位移到 edx 中高 16 位处。
        "movw %1,%%dx" \                // 取[addr+2]处基址低 16 位(位 15-0)➔dx。
        :"=d" (__base) \                // 从而 edx 中含有 32 位的段基地址。
        :"m" (*((addr)+2)), \
         "m" (*((addr)+4)), \
```

```
        "m" (*((addr)+7))); \
__base;})

// 取局部描述符表中 ldt 所指段描述符中的基地址。
#define get_base(ldt) _get_base( ((char *)&(ldt)) )

// 取段选择符 segment 的段长值。
// %0 - 存放段长值(字节数)；%1 - 段选择符 segment。
#define get_limit(segment) ({ \
unsigned long __limit; \
__asm__("lsll %1,%0\n\tincl %0":"=r" (__limit):"r" (segment)); \
__limit;})

#endif

```

11.29.3 其它信息

11.29.3.1 任务状态段信息

31　　　23	15　　　7　　　0	
I/O 映射图基地址(MAP BASE)	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	64
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	局部描述符表(LDT)的选择符	60
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	GS	5C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	FS	58
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DS	54
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS	50
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	CS	4C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ES	48
EDI		44
ESI		40
EBP		3C
ESP		38
EBX		34
EDX		30
ECX		2C
EAX		28
EFLAGS		24
指令指针(EIP)		20
页目录基地址寄存器 CR3（PDBR）		1C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS2	18
ESP2		14
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS1	10
ESP1		0C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS0	08
ESP0		04
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	前一执行任务 TSS 的描述符	00

任务状态段的详细说明请参考附录。这里对其进行简单描述。

CPU 管理任务需要的所有信息被存储于一个特殊类型的段中，任务状态段(task state segment – TSS)。图中显示出执行 80386 任务的 TSS 格式。

TSS 中的字段可以分为两类：

1. CPU 在进行任务切换时更新的动态信息集。这些字段有：
   - o 通用寄存器（EAX，ECX，EDX，EBX，ESP，EBP，ESI，EDI）；
   - o 段寄存器（ES, CS, SS, DS, FS, GS）；
   - o 标志寄存器（EIP）；
   - o 指令指针（EIP）；

   前一个执行任务的 TSS 的选择符（仅当返回时才更新）。
2. CPU 读取但不会更改的静态信息集。这些字段有：
   - o 任务的 LDT 的选择符；
   - o 含有任务页目录基地址的寄存器（PDBR）；
   - o 特权级 0-2 的堆栈指针；
   - o 当任务进行切换时导致 CPU 产生一个调试(debug)异常的 T-比特位（调试跟踪位）；
   - o I/O 比特位图基地址（其长度上限就是 TSS 的长度上限，在 TSS 描述符中说明）。

任务状态段可以存放在线形空间的任何地方。与其它各类段相似，任务状态段也是由描述符来定义的。当前正在执行任务的 TSS 是由任务寄存器（TR）来指示的。指令 LTR 和 STR 用来修改和读取任务寄存器中的选择符（任务寄存器的可见部分）。

I/O 比特位图中的每 1 比特对应 1 个 I/O 端口。比如端口 41 的比特位就是 I/O 位图基地址+5，位偏移 1 处。在保护模式中，当遇到 1 个 I/O 指令时(IN, INS, OUT, OUTS)，CPU 首先就会检查当前特权级是否小于标志寄存器的 IOPL，如果这个条件满足，就执行该 I/O 操作。如果不满足，那么 CPU 就会检查 TSS 中的 I/O 比特位图。如果相应比特位是置位的，就会产生一般保护性异常，否则就会执行该 I/O 操作。

11.29.3.2 段描述符

参见附录。

11.30 sys.h 文件

11.30.1 功能描述

sys.h 头文件列出了内核中所有系统调用函数的原型，以及系统调用函数指针表。

11.30.2 代码注释

列表 linux/include/linux/sys.h 文件

```
extern int sys_setup();          // 系统启动初始化设置函数。     (kernel/blk_drv/hd.c,71)
extern int sys_exit();           // 程序退出。                   (kernel/exit.c, 137)
```

```
extern int sys_fork();          // 创建进程。                  (kernel/system_call.s, 208)
extern int sys_read();          // 读文件。                    (fs/read_write.c, 55)
extern int sys_write();         // 写文件。                    (fs/read_write.c, 83)
extern int sys_open();          // 打开文件。                  (fs/open.c, 138)
extern int sys_close();         // 关闭文件。                  (fs/open.c, 192)
extern int sys_waitpid();       // 等待进程终止。              (kernel/exit.c, 142)
extern int sys_creat();         // 创建文件。                  (fs/open.c, 187)
extern int sys_link();          // 创建一个文件的硬连接。      (fs/namei.c, 721)
extern int sys_unlink();        // 删除一个文件名(或删除文件)。(fs/namei.c, 663)
extern int sys_execve();        // 执行程序。                  (kernel/system_call.s, 200)
extern int sys_chdir();         // 更改当前目录。              (fs/open.c, 75)
extern int sys_time();          // 取当前时间。                (kernel/sys.c, 102)
extern int sys_mknod();         // 建立块/字符特殊文件。       (fs/namei.c, 412)
extern int sys_chmod();         // 修改文件属性。              (fs/open.c, 105)
extern int sys_chown();         // 修改文件宿主和所属组。      (fs/open.c, 121)
extern int sys_break();         //                             (-kernel/sys.c, 21)
extern int sys_stat();          // 使用路径名取文件的状态信息。(fs/stat.c, 36)
extern int sys_lseek();         // 重新定位读/写文件偏移。     (fs/read_write.c, 25)
extern int sys_getpid();        // 取进程 id。                 (kernel/sched.c, 348)
extern int sys_mount();         // 安装文件系统。              (fs/super.c, 200)
extern int sys_umount();        // 卸载文件系统。              (fs/super.c, 167)
extern int sys_setuid();        // 设置进程用户 id。           (kernel/sys.c, 143)
extern int sys_getuid();        // 取进程用户 id。             (kernel/sched.c, 358)
extern int sys_stime();         // 设置系统时间日期。          (-kernel/sys.c, 148)
extern int sys_ptrace();        // 程序调试。                  (-kernel/sys.c, 26)
extern int sys_alarm();         // 设置报警。                  (kernel/sched.c, 338)
extern int sys_fstat();         // 使用文件句柄取文件的状态信息。(fs/stat.c, 47)
extern int sys_pause();         // 暂停进程运行。              (kernel/sched.c, 144)
extern int sys_utime();         // 改变文件的访问和修改时间。  (fs/open.c, 24)
extern int sys_stty();          // 修改终端行设置。            (-kernel/sys.c, 31)
extern int sys_gtty();          // 取终端行设置信息。          (-kernel/sys.c, 36)
extern int sys_access();        // 检查用户对一个文件的访问权限。(fs/open.c, 47)
extern int sys_nice();          // 设置进程执行优先权。        (kernel/sched.c, 378)
extern int sys_ftime();         // 取日期和时间。              (-kernel/sys.c,16)
extern int sys_sync();          // 同步高速缓冲与设备中数据。  (fs/buffer.c, 44)
extern int sys_kill();          // 终止一个进程。              (kernel/exit.c, 60)
extern int sys_rename();        // 更改文件名。                (-kernel/sys.c, 41)
extern int sys_mkdir();         // 创建目录。                  (fs/namei.c, 463)
extern int sys_rmdir();         // 删除目录。                  (fs/namei.c, 587)
extern int sys_dup();           // 复制文件句柄。              (fs/fcntl.c, 42)
extern int sys_pipe();          // 创建管道。                  (fs/pipe.c, 71)
extern int sys_times();         // 取运行时间。                (kernel/sys.c, 156)
extern int sys_prof();          // 程序执行时间区域。          (-kernel/sys.c, 46)
extern int sys_brk();           // 修改数据段长度。            (kernel/sys.c, 168)
extern int sys_setgid();        // 设置进程组 id。             (kernel/sys.c, 72)
extern int sys_getgid();        // 取进程组 id。               (kernel/sched.c, 368)
extern int sys_signal();        // 信号处理。                  (kernel/signal.c, 48)
extern int sys_geteuid();       // 取进程有效用户 id。         (kenrl/sched.c, 363)
extern int sys_getegid();       // 取进程有效组 id。           (kenrl/sched.c, 373)
extern int sys_acct();          // 进程记帐。                  (-kernel/sys.c, 77)
extern int sys_phys();          //                             (-kernel/sys.c, 82)
extern int sys_lock();          //                             (-kernel/sys.c, 87)
extern int sys_ioctl();         // 设备控制。                  (fs/ioctl.c, 30)
```

```
extern int sys_fcntl();          // 文件句柄操作。                 (fs/fcntl.c, 47)
extern int sys_mpx();            //                               (-kernel/sys.c, 92)
extern int sys_setpgid();        // 设置进程组 id。                 (kernel/sys.c, 181)
extern int sys_ulimit();         //                               (-kernel/sys.c, 97)
extern int sys_uname();          // 显示系统信息。                 (kernel/sys.c, 216)
extern int sys_umask();          // 取默认文件创建属性码。           (kernel/sys.c, 230)
extern int sys_chroot();         // 改变根系统。                   (fs/open.c, 90)
extern int sys_ustat();          // 取文件系统信息。               (fs/open.c, 19)
extern int sys_dup2();           // 复制文件句柄。                 (fs/fcntl.c, 36)
extern int sys_getppid();        // 取父进程 id。                   (kernel/sched.c, 353)
extern int sys_getpgrp();        // 取进程组 id，等于 getpgid(0)。(kernel/sys.c, 201)
extern int sys_setsid();         // 在新会话中运行程序。             (kernel/sys.c, 206)
extern int sys_sigaction();      // 改变信号处理过程。               (kernel/signal.c, 63)
extern int sys_sgetmask();       // 取信号屏蔽码。                 (kernel/signal.c, 15)
extern int sys_ssetmask();       // 设置信号屏蔽码。               (kernel/signal.c, 20)
extern int sys_setreuid();       // 设置真实与/或有效用户 id。      (kernel/sys.c,118)
extern int sys_setregid();       // 设置真实与/或有效组 id。        (kernel/sys.c, 51)

// 系统调用函数指针表。用于系统调用中断处理程序(int 0x80)，作为跳转表。
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
sys_setreuid,sys_setregid };

```

11.31 tty.h 文件

11.31.1 功能描述

11.31.2 代码注释

列表 linux/include/linux/tty.h 文件

```
/*
 * 'tty.h' defines some structures used by tty_io.c and some defines.
 *
 * NOTE! Don't touch this without checking that nothing in rs_io.s or
 * con_io.s breaks. Some constants are hardwired into the system (mainly
 * offsets into 'tty_queue'
 */
```

```
/*
 * 'tty.h'中定义了 tty_io.c 程序使用的某些结构和其它一些定义。
 *
 * 注意！在修改这里的定义时，一定要检查 rs_io.s 或 con_io.s 程序中不会出现问题。
 * 在系统中有些常量是直接写在程序中的（主要是一些 tty_queue 中的偏移值）。
 */
#ifndef _TTY_H
#define _TTY_H

#include <termios.h>        // 终端输入输出函数头文件。主要定义控制异步通信口的终端接口。

#define TTY_BUF_SIZE 1024        // tty 缓冲区大小。

// tty 等待队列数据结构。
struct tty_queue {
        unsigned long data;            // 等待队列缓冲区中当前数据指针字符数[??]）。
                                       // 对于串口终端，则存放串行端口地址。
        unsigned long head;            // 缓冲区中数据头指针。
        unsigned long tail;            // 缓冲区中数据尾指针。
        struct task_struct * proc_list;  // 等待进程列表。
        char buf[TTY_BUF_SIZE];        // 队列的缓冲区。
};

// 以下定义了 tty 等待队列中缓冲区操作宏函数。（tail 在前，head 在后）。
// a 缓冲区指针前移 1 字节，并循环。
#define INC(a) ((a) = ((a)+1) & (TTY_BUF_SIZE-1))
// a 缓冲区指针后退 1 字节，并循环。
#define DEC(a) ((a) = ((a)-1) & (TTY_BUF_SIZE-1))
// 清空指定队列的缓冲区。
#define EMPTY(a) ((a).head == (a).tail)
// 缓冲区还可存放字符的长度（空闲区长度）。
#define LEFT(a) (((a).tail-(a).head-1)&(TTY_BUF_SIZE-1))
// 缓冲区中最后一个位置。
#define LAST(a) ((a).buf[(TTY_BUF_SIZE-1)&((a).head-1)])
// 缓冲区满（如果为 1 的话）。
#define FULL(a) (!LEFT(a))
// 缓冲区中已存放字符的长度。
#define CHARS(a) (((a).head-(a).tail)&(TTY_BUF_SIZE-1))
// 从 queue 队列项缓冲区中取一字符(从 tail 处，并且 tail+=1)。
#define GETCH(queue,c) \
(void)({c=(queue).buf[(queue).tail];INC((queue).tail);})
// 往 queue 队列项缓冲区中放置一字符（在 head 处，并且 head+=1）。
#define PUTCH(c,queue) \
(void)({(queue).buf[(queue).head]=(c);INC((queue).head);})

// 判断终端键盘字符类型。
#define INTR_CHAR(tty) ((tty)->termios.c_cc[VINTR])      // 中断符。
#define QUIT_CHAR(tty) ((tty)->termios.c_cc[VQUIT])      // 退出符。
#define ERASE_CHAR(tty) ((tty)->termios.c_cc[VERASE])    // 削除符。
#define KILL_CHAR(tty) ((tty)->termios.c_cc[VKILL])      // 终止符。
#define EOF_CHAR(tty) ((tty)->termios.c_cc[VEOF])        // 文件结束符。
#define START_CHAR(tty) ((tty)->termios.c_cc[VSTART])    // 开始符。
```

```
#define STOP_CHAR(tty) ((tty)->termios.c_cc[VSTOP])     // 结束符。
#define SUSPEND_CHAR(tty) ((tty)->termios.c_cc[VSUSP])  // 挂起符。

// tty 数据结构。
struct tty_struct {
        struct termios termios;                     // 终端 io 属性和控制字符数据结构。
        int pgrp;                                   // 所属进程组。
        int stopped;                                // 停止标志。
        void (*write)(struct tty_struct * tty);     // tty 写函数指针。
        struct tty_queue read_q;                    // tty 读队列。
        struct tty_queue write_q;                   // tty 写队列。
        struct tty_queue secondary;                 // tty 辅助队列(存放规范模式字符序列)，
        };                                          // 可称为规范(熟)模式队列。

extern struct tty_struct tty_table[];               // tty 结构数组。

/*      intr=^C         quit=^|         erase=del       kill=^U
        eof=^D          vtime=\0        vmin=\1         sxtc=\0
        start=^Q        stop=^S         susp=^Z         eol=\0
        reprint=^R      discard=^U      werase=^W       lnext=^V
        eol2=\0
*/
/* 中断 intr=^C      退出 quit=^|      删除 erase=del      终止 kill=^U
 * 文件结束 eof=^D   vtime=\0          vmin=\1             sxtc=\0
 * 开始 start=^Q     停止 stop=^S      挂起 susp=^Z        行结束 eol=\0
 * 重显 reprint=^R   丢弃 discard=^U   werase=^W           lnext=^V
 * 行结束 eol2=\0
 */
// 控制字符对应的 ASCII 码值。[8 进制]
#define INIT_C_CC "\003\034\177\025\004\0\1\0\021\023\032\0\022\017\027\026\0"

void rs_init(void);       // 异步串行通信初始化。(kernel/chr_drv/serial.c, 37)
void con_init(void);      // 控制终端初始化。    (kernel/chr_drv/console.c, 617)
void tty_init(void);      // tty 初始化。        (kernel/chr_drv/tty_io.c, 105)

int tty_read(unsigned c, char * buf, int n);   // (kernel/chr_drv/tty_io.c, 230)
int tty_write(unsigned c, char * buf, int n);  // (kernel/chr_drv/tty_io.c, 290)

void rs_write(struct tty_struct * tty);        // (kernel/chr_drv/serial.c, 53)
void con_write(struct tty_struct * tty);       // (kernel/chr_drv/console.c, 445)

void copy_to_cooked(struct tty_struct * tty);  // (kernel/chr_drv/tty_io.c, 145)

#endif
```

11.32 include/sys/目录中的文件

列表 linux/include/sys/目录下的文件

Name	Size	Last modified (GMT)	Description
stat.h	1304 bytes	1991-09-17 15:02:48	m
times.h	200 bytes	1991-09-17 15:03:06	m
types.h	805 bytes	1991-09-17 15:02:55	m
utsname.h	234 bytes	1991-09-17 15:03:23	m
wait.h	560 bytes	1991-09-17 15:06:07	m

11.33 stat.h 文件

11.33.1 功能描述

该头文件说明了函数 stat()返回的数据及其结构类型，以及一些属性操作测试宏、函数原型。

11.33.2 代码注释

列表 linux/include/sys/stat.h 文件

```
#ifndef _SYS_STAT_H
#define _SYS_STAT_H

#include <sys/types.h>

struct stat {
        dev_t   st_dev;      // 含有文件的设备号。
        ino_t   st_ino;      // 文件 i 节点号。
        umode_t st_mode;     // 文件属性（见下面）。
        nlink_t st_nlink;    // 指定文件的连接数。
        uid_t   st_uid;      // 文件的用户(标识)号。
        gid_t   st_gid;      // 文件的组号。
        dev_t   st_rdev;     // 设备号(如果文件是特殊的字符文件或块文件)。
        off_t   st_size;     // 文件大小（字节数）（如果文件是常规文件）。
        time_t  st_atime;    // 上次（最后）访问时间。
```

```
        time_t  st_mtime;   // 最后修改时间。
        time_t  st_ctime;   // 最后节点修改时间。
};

// 以下这些是 st_mode 值的符号名称。
// 文件类型:
#define S_IFMT  00170000    // 文件类型。
#define S_IFREG  0100000    // 常规文件。
#define S_IFBLK  0060000    // 块特殊（设备）文件。
#define S_IFDIR  0040000    // 目录文件。
#define S_IFCHR  0020000    // 字符设备文件。
#define S_IFIFO  0010000    // FIFO 特殊文件。
// 文件属性位:
#define S_ISUID  0004000    // 执行时设置用户 ID（set-user-ID）。
#define S_ISGID  0002000    // 执行时设置组 ID。
#define S_ISVTX  0001000    // 对于目录，受限删除标志。

#define S_ISREG(m)      (((m) & S_IFMT) == S_IFREG)    // 测试是否常规文件。
#define S_ISDIR(m)      (((m) & S_IFMT) == S_IFDIR)    // 是否目录文件。
#define S_ISCHR(m)      (((m) & S_IFMT) == S_IFCHR)    // 是否字符设备文件。
#define S_ISBLK(m)      (((m) & S_IFMT) == S_IFBLK)    // 是否块设备文件。
#define S_ISFIFO(m)     (((m) & S_IFMT) == S_IFIFO)    // 是否 FIFO 特殊文件。

#define S_IRWXU 00700      // 宿主可以读、写、执行/搜索。
#define S_IRUSR 00400      // 宿主读许可。
#define S_IWUSR 00200      // 宿主写许可。
#define S_IXUSR 00100      // 宿主执行/搜索许可。

#define S_IRWXG 00070      // 组成员可以读、写、执行/搜索。
#define S_IRGRP 00040      // 组成员读许可。
#define S_IWGRP 00020      // 组成员写许可。
#define S_IXGRP 00010      // 组成员执行/搜索许可。

#define S_IRWXO 00007      // 其他人读、写、执行/搜索许可。
#define S_IROTH 00004      // 其他人读许可。
#define S_IWOTH 00002      // 其他人写许可。
#define S_IXOTH 00001      // 其他人执行/搜索许可。

extern int chmod(const char *_path, mode_t mode);      // 修改文件属性。
extern int fstat(int fildes, struct stat *stat_buf);  // 取指定文件句柄的文件状态信息。
extern int mkdir(const char *_path, mode_t mode);      // 创建目录。
extern int mkfifo(const char *_path, mode_t mode);     // 创建管道文件。
extern int stat(const char *filename, struct stat *stat_buf); // 取指定文件名的文件状态信息。
extern mode_t umask(mode_t mask);   // 设置属性屏蔽码。

#endif
```

11.34 times.h 文件

11.34.1 功能描述

该头文件中主要定义了文件访问与修改时间结构 tms。它将由 times()函数返回。其中 time_t 是在 sys/types.h 中定义的。还定义了一个函数原型 times()。

11.34.2 代码注释

列表 linux/include/sys/times.h 文件

```
#ifndef _TIMES_H
#define _TIMES_H

#include <sys/types.h>      // 类型头文件。定义了基本的系统数据类型。

struct tms {
        time_t tms_utime;   // 用户使用的 CPU 时间。
        time_t tms_stime;   // 系统（内核）CPU 时间。
        time_t tms_cutime;  // 已终止的子进程使用的用户 CPU 时间。
        time_t tms_cstime;  // 已终止的子进程使用的系统 CPU 时间。
};

extern time_t times(struct tms * tp);

#endif

```

11.35 types.h 文件

11.35.1 功能描述

types.h 头文件中定义了基本的数据类型。所有的类型定义为适当的数学类型长度。另外，size_t 是无符号整数类型，off_t 是扩展的符号整数类型，pid_t 是符号整数类型。

11.35.2 代码注释

列表 linux/include/sys/types.h 文件

```
#ifndef _SYS_TYPES_H
#define _SYS_TYPES_H

#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;         // 用于对象的大小（长度）。
#endif

#ifndef _TIME_T
```

```
#define _TIME_T
typedef long time_t;                // 用于时间（以秒计）。
#endif

#ifndef _PTRDIFF_T
#define _PTRDIFF_T
typedef long ptrdiff_t;
#endif

#ifndef NULL
#define NULL ((void *) 0)
#endif

typedef int pid_t;                  // 用于进程号和进程组号。
typedef unsigned short uid_t;       // 用于用户号（用户标识号）。
typedef unsigned char gid_t;        // 用于组号。
typedef unsigned short dev_t;       // 用于设备号。
typedef unsigned short ino_t;       // 用于文件序列号。
typedef unsigned short mode_t;      // 用于某些文件属性。
typedef unsigned short umode_t;     //
typedef unsigned char nlink_t;      // 用于连接计数。
typedef int daddr_t;
typedef long off_t;                 // 用于文件长度（大小）。
typedef unsigned char u_char;       // 无符号字符类型。
typedef unsigned short ushort;      // 无符号短整数类型。

typedef struct { int quot,rem; } div_t;     // 用于 DIV 操作。
typedef struct { long quot,rem; } ldiv_t;   // 用于长 DIV 操作。

struct ustat {
        daddr_t f_tfree;
        ino_t f_tinode;
        char f_fname[6];
        char f_fpack[6];
};

#endif

```

11.36 utsname.h 文件

11.36.1 功能描述

utsname.h 是系统名称结构头文件。其中定义了结构 utsname 以及函数原型 uname()。POSIX 要求字符数组长度应该是不指定的，但是其中存储的数据需以 null 终止。因此该版内核的 utsname 结构定义不符要求（数组长度都被定义为 9）。

11.36.2 代码注释

列表 linux/include/sys/utsname.h 文件

```
#ifndef _SYS_UTSNAME_H
#define _SYS_UTSNAME_H

#include <sys/types.h>      // 类型头文件。定义了基本的系统数据类型。

struct utsname {
        char sysname[9];    // 本版本操作系统的名称。
        char nodename[9];   // 与实现相关的网络中节点名称。
        char release[9];    // 本实现的当前发行级别。
        char version[9];    // 本次发行的版本级别。
        char machine[9];    // 系统运行的硬件类型名称。
};

extern int uname(struct utsname * utsbuf);

#endif

```

11.37 wait.h 文件

11.37.1 功能描述

该头文件描述了进程等待时信息。包括一些符号常数和 wait()、waitpid()函数原型申明。

11.37.2 代码注释

列表 linux/include/sys/wait.h 文件

```
#ifndef _SYS_WAIT_H
#define _SYS_WAIT_H

#include <sys/types.h>

#define _LOW(v)         ( (v) & 0377)          // 取低字节（8 进制表示）。
#define _HIGH(v)        ( ((v) >> 8) & 0377)   // 取高字节。

/* options for waitpid, WUNTRACED not supported */
/* waitpid 的选项，其中 WUNTRACED 未被支持 */
#define WNOHANG         1       // 如果没有状态也不要挂起，并立刻返回。
#define WUNTRACED       2       // 报告停止执行的子进程状态。

#define WIFEXITED(s)    (!((s)&0xFF)        // 如果子进程正常退出，则为真。
#define WIFSTOPPED(s)   (((s)&0xFF)==0x7F)  // 如果子进程正停止着，则为 true。
#define WEXITSTATUS(s)  (((s)>>8)&0xFF)     // 返回退出状态。
#define WTERMSIG(s)     ((s)&0x7F)          // 返回导致进程终止的信号值（信号量）。
#define WSTOPSIG(s)     (((s)>>8)&0xFF)     // 返回导致进程停止的信号值。
#define WIFSIGNALED(s)  (((unsigned int)(s)-1 & 0xFFFF) < 0xFF) // 如果由于未捕捉到信号
                                                                // 而导致子进程退出则为真。

// wait()和 waitpit()函数允许进程获取与其子进程之一的状态信息。各种选项允许获取已经终止或
```

```
// 停止的子进程状态信息。如果存在两个或两个以上子进程的状态信息，则报告的顺序是不指定的。
// wait()将挂起当前进程，直到其子进程之一退出（终止），或者收到要求终止该进程的信号，
// 或者是需要调用一个信号句柄（信号处理程序）。
// waitpid()挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止该进程的信号，
// 或者是需要调用一个信号句柄（信号处理程序）。
// 如果 pid=-1，options=0，则 waitpid()的作用与 wait()函数一样。否则其行为将随 pid 和 options
// 参数的不同而不同。（参见 kernel/exit.c,142）
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);

#endif

```

第 12 章 库文件(lib)

12.1 概述

Name	Size	Last modified (GMT)	Description
Makefile	2602 bytes	1991-12-02 03:16:05	
_exit.c	198 bytes	1991-10-02 14:16:29	
close.c	131 bytes	1991-10-02 14:16:29	
ctype.c	1202 bytes	1991-10-02 14:16:29	
dup.c	127 bytes	1991-10-02 14:16:29	
errno.c	73 bytes	1991-10-02 14:16:29	
execve.c	170 bytes	1991-10-02 14:16:29	
malloc.c	7469 bytes	1991-12-02 03:15:20	
open.c	389 bytes	1991-10-02 14:16:29	
setsid.c	128 bytes	1991-10-02 14:16:29	
string.c	177 bytes	1991-10-02 14:16:29	
wait.c	253 bytes	1991-10-02 14:16:29	
write.c	160 bytes	1991-10-02 14:16:29	

12.2 Makefile 文件

12.2.1 功能描述

12.2.2 代码注释

列表 linux/lib/Makefile 文件

```
#
# Makefile for some libs needed in the kernel.
#
# Note! Dependencies are done automagically by 'make dep', which also
# removes any old dependencies. DON'T put your own dependencies here
# unless it's something special (ie not a .c file).
#
# 内核需要用到的 libs 库文件程序的 Makefile。
#
# 注意！依赖关系是由'make dep'自动进行的，它也会自动去除原来的依赖信息。不要把你自己的
```

```
  # 依赖关系信息放在这里，除非是特别文件的（也即不是一个.c 文件的信息）。
 8
 9 AR      =gar    # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。
10 AS      =gas    # GNU 的汇编程序。
11 LD      =gld    # GNU 的连接程序。
12 LDFLAGS =-s -x  # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
13 CC      =gcc    # GNU C 语言编译器。
14 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
15         -finline-functions -mstring-insns -nostdinc -I../include
   # C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
   # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
   # 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
   # 单短小的函数代码嵌入调用程序中；-mstring-insns Linus 自己填加的优化选项，以后不再使用；
   # -nostdinc -I../include 不使用默认路径中的包含文件，而使用这里指定目录中的(../include)。

16 CPP     =gcc -E -nostdinc -I../include
   # C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
   # 出设备或指定的输出文件中；-nostdinc -I../include 同前。
17
   # 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
   # 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
   # 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
   # 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。其中$*.s（或$@）是自动目标变量，
   # $<代表第一个先决条件，这里即是符合条件*.c 的文件。
18 .c.s:
19         $(CC) $(CFLAGS) \
20         -S -o $*.s $<
  # 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。22 行是实现该操作的具体命令。
21 .s.o:
22         $(AS) -c -o $*.o $<
23 .c.o:                        # 类似上面，*.c 文件-→*.o 目标文件。不进行连接。
24         $(CC) $(CFLAGS) \
25         -c -o $*.o $<
26
  # 下面定义目标文件变量 OBJS。
27 OBJS  = ctype.o _exit.o open.o close.o errno.o write.o dup.o setsid.o \
28         execve.o wait.o string.o malloc.o
29
  # 在有了先决条件 OBJS 后使用下面的命令连接成目标 lib.a 库文件。
30 lib.a: $(OBJS)
31         $(AR) rcs lib.a $(OBJS)
32         sync
33
   # 下面的规则用于清理工作。当执行'make clean'时，就会执行下面的命令，去除所有编译
   # 连接生成的文件。'rm'是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
34 clean:
35         rm -f core *.o *.a tmp_make
36         for i in *.c;do rm -f `basename $$i .c`.s;done
37
   # 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
   # 使用字符串编辑程序 sed 对 Makefile 文件（即是本文件）进行处理，输出为删除 Makefile
   # 文件中'### Dependencies'行后面的所有行（下面从 45 开始的行），并生成 tmp_make
   # 临时文件（39 行的作用）。然后对 kernel/blk_drv/目录下的每个 C 文件执行 gcc 预处理操作.
```

```
    # -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
    # 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
    # 文件名加上其依赖关系--该源文件中包含的所有头文件列表。把预处理结果都添加到临时
    # 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。
dep:
        sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
        (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,'`" "; \
                $(CPP) -M $$i;done) >> tmp_make
        cp tmp_make Makefile

### Dependencies:
_exit.s _exit.o : _exit.c ../include/unistd.h ../include/sys/stat.h \
  ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
  ../include/utime.h
close.s close.o : close.c ../include/unistd.h ../include/sys/stat.h \
  ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
  ../include/utime.h
ctype.s ctype.o : ctype.c ../include/ctype.h
dup.s dup.o : dup.c ../include/unistd.h ../include/sys/stat.h \
  ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
  ../include/utime.h
errno.s errno.o : errno.c
execve.s execve.o : execve.c ../include/unistd.h ../include/sys/stat.h \
  ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
  ../include/utime.h
malloc.s malloc.o : malloc.c ../include/linux/kernel.h ../include/linux/mm.h \
  ../include/asm/system.h
open.s open.o : open.c ../include/unistd.h ../include/sys/stat.h \
  ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
  ../include/utime.h ../include/stdarg.h
setsid.s setsid.o : setsid.c ../include/unistd.h ../include/sys/stat.h \
  ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
  ../include/utime.h
string.s string.o : string.c ../include/string.h
wait.s wait.o : wait.c ../include/unistd.h ../include/sys/stat.h \
  ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
  ../include/utime.h ../include/sys/wait.h
write.s write.o : write.c ../include/unistd.h ../include/sys/stat.h \
  ../include/sys/types.h ../include/sys/times.h ../include/sys/utsname.h \
  ../include/utime.h
```

12.3 _exit.c 程序

12.3.1 功能描述

12.3.2 代码注释

列表 linux/lib/_exit.c 程序

```
/*
 *  linux/lib/_exit.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define __LIBRARY__             // 定义一个符号常量，见下行说明。
#include <unistd.h>             // Linux 标准头文件。定义了各种符号常数和类型，并申明了各种函数。
                                // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()等。

//// 内核使用的程序(退出)终止函数。
// 直接调用系统中断 int 0x80，功能号__NR_exit。
// 参数：exit_code - 退出码。
volatile void _exit(int exit_code)
{
// %0 - eax(系统调用号__NR_exit)；%1 - ebx(退出码 exit_code)。
        __asm__("int $0x80"::"a" (__NR_exit),"b" (exit_code));
}

```

12.3.3 相关信息

参见 include/unistd.h 中的说明。

12.4 close.c 程序

12.4.1 功能描述

12.4.2 代码注释

列表 linux/lib/close.c 程序

```
/*
 *  linux/lib/close.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define __LIBRARY__
#include <unistd.h>             // Linux 标准头文件。定义了各种符号常数和类型，并申明了各种函数。
                                // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()等。

// 关闭文件函数。
// 下面该调用宏函数对应：int close(int fd)。直接调用了系统中断 int 0x80，参数是__NR_close。
// 其中 fd 是文件描述符。
_syscall1(int,close,int,fd)

```

12.5 ctype.c 程序

12.5.1 功能描述

12.5.2 代码注释

列表 linux/lib/ctype.c 程序

```
/*
 *  linux/lib/ctype.c
 *
 *  (C) 1991  Linus Torvalds
 */

#include <ctype.h>          // 字符类型头文件。定义了一些有关字符类型判断和转换的宏。

char _ctmp;                 // 一个临时字符变量，供 ctype.h 文件中转换字符宏函数使用。
// 字符特性数组(表)，定义了各个字符对应的属性，这些属性类型(如_C 等)在 ctype.h 中定义。
// 用于判断字符是控制字符(_C)、大写字符(_U)、小写字符(_L)等所属类型。
unsigned char _ctype[] = {0x00,                 /* EOF */
_C,_C,_C,_C,_C,_C,_C,_C,                        /* 0-7 */
_C,_C|_S,_C|_S,_C|_S,_C|_S,_C|_S,_C,_C,         /* 8-15 */
_C,_C,_C,_C,_C,_C,_C,_C,                        /* 16-23 */
_C,_C,_C,_C,_C,_C,_C,_C,                        /* 24-31 */
_S|_SP,_P,_P,_P,_P,_P,_P,_P,                    /* 32-39 */
_P,_P,_P,_P,_P,_P,_P,_P,                        /* 40-47 */
_D,_D,_D,_D,_D,_D,_D,_D,                        /* 48-55 */
_D,_D,_P,_P,_P,_P,_P,_P,                        /* 56-63 */
_P,_U|_X,_U|_X,_U|_X,_U|_X,_U|_X,_U|_X,_U,      /* 64-71 */
_U,_U,_U,_U,_U,_U,_U,_U,                        /* 72-79 */
_U,_U,_U,_U,_U,_U,_U,_U,                        /* 80-87 */
_U,_U,_U,_P,_P,_P,_P,_P,                        /* 88-95 */
_P,_L|_X,_L|_X,_L|_X,_L|_X,_L|_X,_L|_X,_L,      /* 96-103 */
_L,_L,_L,_L,_L,_L,_L,_L,                        /* 104-111 */
_L,_L,_L,_L,_L,_L,_L,_L,                        /* 112-119 */
_L,_L,_L,_P,_P,_P,_P,_C,                        /* 120-127 */
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                /* 128-143 */
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                /* 144-159 */
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                /* 160-175 */
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                /* 176-191 */
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                /* 192-207 */
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                /* 208-223 */
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,                /* 224-239 */
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};               /* 240-255 */
```

12.6 dup.c 程序

12.6.1 功能描述

该程序包括一个创建文件描述符拷贝的函数 dup()。在成功返回之后，新的和原来的描述符可以交替使用。它们共享锁定、文件读写指针以及文件标志。例如，如果文件读写位置指针被其中一个描述符使用 lseek()修改过之后，则对于另一个描述符来讲，文件读写指针也被改变。该函数使用数值最小的未使用描述符来建立新描述符。但是这两个描述符并不共享执行时关闭标志(close-on-exec)。

12.6.2 代码注释

列表 linux/lib/dup.c 程序

```
/*
 *  linux/lib/dup.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define __LIBRARY__
#include <unistd.h>         // Linux 标准头文件。定义了各种符号常数和类型，并申明了各种函数。
                            // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()等。

//// 复制文件描述符函数。
// 下面该调用宏函数对应：int dup(int fd)。直接调用了系统中断 int 0x80，参数是__NR_dup。
// 其中 fd 是文件描述符。
_syscall1(int,dup,int,fd)

```

12.7 errno.c 程序

12.7.1 功能描述

该程序仅定义了一个出错号变量 errno。用于在函数调用失败时存放出错号。

12.7.2 代码注释

列表 linux/lib/errno.c 程序

```
/*
 *  linux/lib/errno.c
 *
 *  (C) 1991  Linus Torvalds
 */

int errno;
```

12.8 execve.c 程序

12.8.1 功能描述

12.8.2 代码注释

列表 linux/lib/execve.c 程序

```
/*
 *  linux/lib/execve.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define __LIBRARY__
#include <unistd.h>          // Linux 标准头文件。定义了各种符号常数和类型，并申明了各种函数。
                             // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()
等。

//// 加载并执行子进程(其它程序)函数。
// 下面该调用宏函数对应：int execve(const char * file, char ** argv, char ** envp)。
// 参数：file - 被执行程序文件名；argv - 命令行参数指针数组；envp - 环境变量指针数组。
// 直接调用了系统中断 int 0x80，参数是__NR_execve。参见 include/unistd.h 和 fs/exec.c 程序。
_syscall3(int,execve,const char *,file,char **,argv,char **,envp)

```

12.9 malloc.c 程序

12.9.1 功能描述

该程序中主要包括内存分配函数 malloc()。该函数使用了存储桶(bucket)的原理对分配的内存进行管理。基本思想是对不同请求的内存块大小(长度)，使用存储桶目录(下面简称目录)分别进行处理。比如对于请求内存块的长度在 32 字节或 32 字节以下但大于 16 字节时，就使用存储桶目录第二项对应的存储桶描述符链表分配内存块。其基本结构示意图见下图所示。该函数目前一次所能分配的最大内存长度是一个内存页面，即 4096 字节。

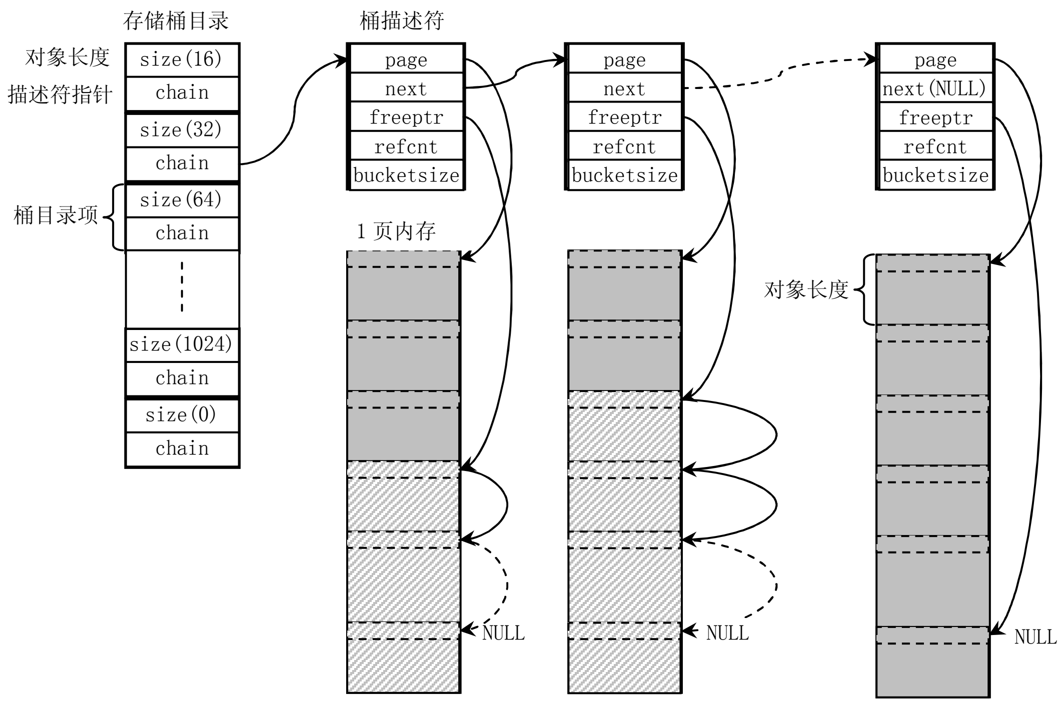


图 使用存储桶原理进行内存分配管理的结构示意图

在第一次调用 malloc()函数时，首先要建立一个页面的空闲存储桶描述符(下面简称描述符)链表，其中存放着还未使用或已经使用完毕而收回的描述符。该链表结构示意图见下图所示。其中 free_bucket_desc 是链表头指针。从链表中取出或放入一个描述符都是从链表头开始操作。当取出一个描述符时，就将链表头指针所指向的头一个描述符取出；当释放一个空闲描述符时也是将其放在链表头处。

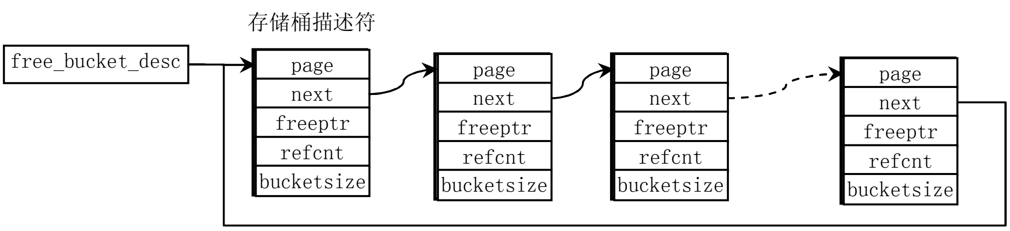


图 空闲存储桶描述符链表结构示意图

malloc()函数执行的基本步骤如下：

1. 首先搜索目录，寻找适合请求内存块大小的目录项对应的描述符链表。当目录项的对象字节长度大于请求的字节长度，就算找到了相应的目录项。如果搜索完整个目录都没有找到合适的目录项，则说明用户请求的内存块太大。
2. 在目录项对应的描述符链表中查找具有空闲空间的描述符。如果某个描述符的空闲内存指针 freeptr 不为 NULL，则表示找到了相应的描述符。如果没有找到具有空闲空间的描述符，那么我们就需要新建一个描述符。新建描述符的过程如下：
   a. 如果空闲描述符链表头指针还是 NULL 的话，说明是第一次调用 malloc()函数，此时需要 init_bucket_desc()来创建空闲描述符链表。

b. 然后从空闲描述符链表头处取一个描述符，初始化该描述符，令其对象引用计数为 0，对象大小等于对应目录项指定对象的长度值，并申请一内存页面，让描述符的页面指针 page 指向该内存页，描述符的空闲内存指针 freeptr 也指向页开始位置。
c. 对该内存页面根据本目录项所用对象长度进行页面初始化，建立所有对象的一个链表。也即每个对象的头部都存放一个指向下一个对象的指针，最后一个对象的开始处存放一个 NULL 指针值。
d. 然后将该描述符插入到对应目录项的描述符链表开始处。

3. 将该描述符的空闲内存指针 freeptr 复制为返回给用户的内存指针，然后调整该 freeptr 指向描述符对应内存页面中下一个空闲对象位置，并使该描述符引用计数值增 1。

free_s()函数用于回收用户释放的内存块。基本原理是首先根据该内存块的地址换算出该内存块对应页面的地址(用页面长度进行模运算)，然后搜索目录中的所有描述符，找到对应该页面的描述符。将该释放的内存块链入 freeptr 所指向的空闲对象链表中，并将描述符的对象引用计数值减 1。如果引用计数值此时等于零，则表示该描述符对应的页面已经完全空出，可以释放该内存页面并将该描述符收回到空闲描述符链表中。

12.9.2 代码注释

列表 linux/lib/malloc.c 程序

```
/*
 * malloc.c --- a general purpose kernel memory allocator for Linux.
 *
 * Written by Theodore Ts'o (tytso@mit.edu), 11/29/91
 *
 * This routine is written to be as fast as possible, so that it
 * can be called from the interrupt level.
 *
 * Limitations: maximum size of memory we can allocate using this routine
 *      is 4k, the size of a page in Linux.
 *
 * The general game plan is that each page (called a bucket) will only hold
 * objects of a given size.  When all of the object on a page are released,
 * the page can be returned to the general free pool.  When malloc() is
 * called, it looks for the smallest bucket size which will fulfill its
 * request, and allocate a piece of memory from that bucket pool.
 *
 * Each bucket has as its control block a bucket descriptor which keeps
 * track of how many objects are in use on that page, and the free list
 * for that page.  Like the buckets themselves, bucket descriptors are
 * stored on pages requested from get_free_page().  However, unlike buckets,
 * pages devoted to bucket descriptor pages are never released back to the
 * system.  Fortunately, a system should probably only need 1 or 2 bucket
 * descriptor pages, since a page can hold 256 bucket descriptors (which
 * corresponds to 1 megabyte worth of bucket pages.)  If the kernel is using
 * that much allocated memory, it's probably doing something wrong.  :-)
 *
 * Note: malloc() and free() both call get_free_page() and free_page()
 *      in sections of code where interrupts are turned off, to allow
 *      malloc() and free() to be safely called from an interrupt routine.
```

```
 *      (We will probably need this functionality when networking code,
 *      particularily things like NFS, is added to Linux.)  However, this
 *      presumes that get_free_page() and free_page() are interrupt-level
 *      safe, which they may not be once paging is added.  If this is the
 *      case, we will need to modify malloc() to keep a few unused pages
 *      "pre-allocated" so that it can safely draw upon those pages if
 *      it is called from an interrupt routine.
 *
 *      Another concern is that get_free_page() should not sleep; if it
 *      does, the code is carefully ordered so as to avoid any race
 *      conditions.  The catch is that if malloc() is called re-entrantly,
 *      there is a chance that unecessary pages will be grabbed from the
 *      system.  Except for the pages for the bucket descriptor page, the
 *      extra pages will eventually get released back to the system, though,
 *      so it isn't all that bad.
 */

/*
 *      malloc.c - Linux 的通用内核内存分配函数。
 *
 * 由 Theodore Ts'o 编制 (tytso@mit.edu), 11/29/91
 *
 * 该函数被编写成尽可能地快，从而可以从中断层调用此函数。
 *
 * 限制：使用该函数一次所能分配的最大内存是 4k，也即 Linux 中内存页面的大小。
 *
 * 编写该函数所遵循的一般规则是每页(被称为一个存储桶)仅分配所要容纳对象的大小。
 * 当一页上的所有对象都释放后，该页就可以返回通用空闲内存池。当 malloc()被调用
 * 时，它会寻找满足要求的最小的存储桶，并从该存储桶中分配一块内存。
 *
 * 每个存储桶都有一个作为其控制用的存储桶描述符，其中记录了页面上有多少对象正被
 * 使用以及该页上空闲内存的列表。就象存储桶自身一样，存储桶描述符也是存储在使用
 * get_free_page()申请到的页面上的，但是与存储桶不同的是，桶描述符所占用的页面
 * 将不再会释放给系统。幸运的是一个系统大约只需要 1 到 2 页的桶描述符页面，因为一
 * 个页面可以存放 256 个桶描述符(对应 1MB 内存的存储桶页面)。如果系统为桶描述符分
 * 配了许多内存，那么肯定系统什么地方出了问题☺。
 *
 * 注意！malloc()和 free()两者关闭了中断的代码部分都调用了 get_free_page()和
 *         free_page()函数，以使 malloc()和 free()可以安全地被从中断程序中调用
 *         (当网络代码，尤其是 NFS 等被加入到 Linux 中时就可能需要这种功能)。但前
 *         提是假设 get_free_page()和 free_page()是可以安全地在中断级程序中使用的，
 *         这在一旦加入了分页处理之后就可能不是安全的。如果真是这种情况，那么我们
 *         就需要修改 malloc()来“预先分配”几页不用的内存，如果 malloc()和 free()
 *         被从中断程序中调用时就可以安全地使用这些页面。
 *
 *         另外需要考虑到的是 get_free_page()不应该睡眠；如果会睡眠的话，则为了防止
 *         任何竞争条件，代码需要仔细地安排顺序。 关键在于如果 malloc()是可以重入地
 *         被调用的话，那么就会存在不必要的页面被从系统中取走的机会。除了用于桶描述
 *         符的页面，这些额外的页面最终会释放给系统，所以并不是象想象的那样不好。
 */

#include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
#include <linux/mm.h>      // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
```

```
#include <asm/system.h>    // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

// 存储桶描述符结构。
struct bucket_desc {    /* 16 bytes */
        void                    *page;          // 该桶描述符对应的内存页面指针。
        struct bucket_desc      *next;          // 下一个描述符指针。
        void                    *freeptr;       // 指向本桶中空闲内存位置的指针。
        unsigned short          refcnt;         // 引用计数。
        unsigned short          bucket_size;    // 本描述符对应存储桶的大小。
};

// 存储桶描述符目录结构。
struct _bucket_dir {    /* 8 bytes */
        int                     size;       // 该存储桶的大小(字节数)。
        struct bucket_desc      *chain;     // 该存储桶目录项的桶描述符链表指针。
};

/*
 * The following is the where we store a pointer to the first bucket
 * descriptor for a given size.
 *
 * If it turns out that the Linux kernel allocates a lot of objects of a
 * specific size, then we may want to add that specific size to this list,
 * since that will allow the memory to be allocated more efficiently.
 * However, since an entire page must be dedicated to each specific size
 * on this list, some amount of temperance must be exercised here.
 *
 * Note that this list *must* be kept in order.
 */
/*
 * 下面是我们存放第一个给定大小存储桶描述符指针的地方。
 *
 * 如果 Linux 内核分配了许多指定大小的对象，那么我们就希望将该指定的大小加到
 * 该列表(链表)中，因为这样可以使内存的分配更有效。但是，因为一页完整内存页面
 * 必须用于列表中指定大小的所有对象，所以需要做总数方面的测试操作。
 */
// 存储桶目录列表(数组)。
struct _bucket_dir bucket_dir[] = {
        { 16,   (struct bucket_desc *) 0},    // 16 字节长度的内存块。
        { 32,   (struct bucket_desc *) 0},    // 32 字节长度的内存块。
        { 64,   (struct bucket_desc *) 0},    // 64 字节长度的内存块。
        { 128,  (struct bucket_desc *) 0},    // 128 字节长度的内存块。
        { 256,  (struct bucket_desc *) 0},    // 256 字节长度的内存块。
        { 512,  (struct bucket_desc *) 0},    // 512 字节长度的内存块。
        { 1024, (struct bucket_desc *) 0},    // 1024 字节长度的内存块。
        { 2048, (struct bucket_desc *) 0},    // 2048 字节长度的内存块。
        { 4096, (struct bucket_desc *) 0},    // 4096 字节(1 页)内存。
        { 0,    (struct bucket_desc *) 0}};   /* End of list marker */

/*
 * This contains a linked list of free bucket descriptor blocks
 */
/*
```

```
 * 下面是含有空闲桶描述符内存块的链表。
 */
struct bucket_desc *free_bucket_desc = (struct bucket_desc *) 0;

/*
 * This routine initializes a bucket description page.
 */
/*
 * 下面的子程序用于初始化一页桶描述符页面。
 */
//// 初始化桶描述符。
// 建立空闲桶描述符链表，并让 free_bucket_desc 指向第一个空闲桶描述符。
static inline void init_bucket_desc()
{
        struct bucket_desc *bdesc, *first;
        int     i;

// 申请一页内存，用于存放桶描述符。如果失败，则显示初始化桶描述符时内存不够出错信息，死机。
        first = bdesc = (struct bucket_desc *) get_free_page();
        if (!bdesc)
                panic("Out of memory in init_bucket_desc()");
// 首先计算一页内存中可存放的桶描述符数量，然后对其建立单向连接指针。
        for (i = PAGE_SIZE/sizeof(struct bucket_desc); i > 1; i--) {
                bdesc->next = bdesc+1;
                bdesc++;
        }
        /*
         * This is done last, to avoid race conditions in case
         * get_free_page() sleeps and this routine gets called again....
         */
        /*
         * 这是在最后处理的，目的是为了避免在 get_free_page()睡眠时该子程序又被
         * 调用而引起的竞争条件。
         */
// 将空闲桶描述符指针 free_bucket_desc 加入链表中。
        bdesc->next = free_bucket_desc;
        free_bucket_desc = first;
}

//// 分配动态内存函数。
// 参数：len - 请求的内存块长度。
// 返回：指向被分配内存的指针。如果失败则返回 NULL。
void *malloc(unsigned int len)
{
        struct _bucket_dir      *bdir;
        struct bucket_desc      *bdesc;
        void                    *retval;

        /*
         * First we search the bucket_dir to find the right bucket change
         * for this request.
         */
        /*
```

```
         * 首先我们搜索存储桶目录 bucket_dir 来寻找适合请求的桶大小。
         */
// 搜索存储桶目录，寻找适合申请内存块大小的桶描述符链表。如果目录项的桶字节数大于请求的字节
// 数，就找到了对应的桶目录项。
        for (bdir = bucket_dir; bdir->size; bdir++)
                if (bdir->size >= len)
                        break;
// 如果搜索完整个目录都没有找到合适大小的目录项，则表明所请求的内存块大小太大，超出了该
// 程序的分配限制(最长为 1 个页面)。于是显示出错信息，死机。
        if (!bdir->size) {
                printk("malloc called with impossibly large argument (%d)\n",
                        len);
                panic("malloc: bad arg");
        }
        /*
         * Now we search for a bucket descriptor which has free space
         */
        /*
         * 现在我们来搜索具有空闲空间的桶描述符。
         */
        cli();  /* Avoid race conditions */   /* 为了避免出现竞争条件，首先关中断 */
// 搜索对应桶目录项中描述符链表，查找具有空闲空间的桶描述符。如果桶描述符的空闲内存指针
// freeptr 不为空，则表示找到了相应的桶描述符。
        for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next)
                if (bdesc->freeptr)
                        break;
        /*
         * If we didn't find a bucket with free space, then we'll
         * allocate a new one.
         */
        /*
         * 如果没有找到具有空闲空间的桶描述符，那么我们就要新建立一个该目录项的描述符。
         */
        if (!bdesc) {
                char            *cp;
                int             i;

// 若 free_bucket_desc 还为空时，表示第一次调用该程序，则对描述符链表进行初始化。
// free_bucket_desc 指向第一个空闲桶描述符。
                if (!free_bucket_desc)
                        init_bucket_desc();
// 取 free_bucket_desc 指向的空闲桶描述符，并让 free_bucket_desc 指向下一个空闲桶描述符。
                bdesc = free_bucket_desc;
                free_bucket_desc = bdesc->next;
// 初始化该新的桶描述符。令其引用数量等于 0；桶的大小等于对应桶目录的大小；申请一内存页面，
// 让描述符的页面指针 page 指向该页面；空闲内存指针也指向该页开头，因为此时全为空闲。
                bdesc->refcnt = 0;
                bdesc->bucket_size = bdir->size;
                bdesc->page = bdesc->freeptr = (void *) cp = get_free_page();
// 如果申请内存页面操作失败，则显示出错信息，死机。
                if (!cp)
                        panic("Out of memory in kernel malloc()");
                /* Set up the chain of free objects */
```

```
                    /* 在该页空闲内存中建立空闲对象链表 */
// 以该桶目录项指定的桶大小为对象长度，对该页内存进行划分，并使每个对象的开始 4 字节设置
// 成指向下一对象的指针。
                    for (i=PAGE_SIZE/bdir->size; i > 1; i--) {
                            *((char **) cp) = cp + bdir->size;
                            cp += bdir->size;
                    }
// 最后一个对象开始处的指针设置为 0(NULL)。
// 然后让该桶描述符的下一描述符指针字段指向对应桶目录项指针 chain 所指的描述符，而桶目录的
// chain 指向该桶描述符，也即将该描述符插入到描述符链链头处。
                    *((char **) cp) = 0;
                    bdesc->next = bdir->chain; /* OK, link it in! */   /* OK，将其链入！*/
                    bdir->chain = bdesc;
            }
// 返回指针即等于该描述符对应页面的当前空闲指针。然后调整该空闲空间指针指向下一个空闲对象，
// 并使描述符中对应页面中对象引用计数增 1。
            retval = (void *) bdesc->freeptr;
            bdesc->freeptr = *((void **) retval);
            bdesc->refcnt++;
// 最后开放中断，并返回指向空闲内存对象的指针。
            sti();  /* OK, we're safe again */   /* OK，现在我们又安全了*/
            return(retval);
}

/*
 * Here is the free routine.  If you know the size of the object that you
 * are freeing, then free_s() will use that information to speed up the
 * search for the bucket descriptor.
 *
 * We will #define a macro so that "free(x)" is becomes "free_s(x, 0)"
 */
/*
 * 下面是释放子程序。如果你知道释放对象的大小，则 free_s()将使用该信息加速
 * 搜寻对应桶描述符的速度。
 *
 * 我们将定义一个宏，使得"free(x)"成为"free_s(x, 0)"。
 */
//// 释放存储桶对象。
// 参数：obj - 对应对象指针；size - 大小。
void free_s(void *obj, int size)
{
            void            *page;
            struct _bucket_dir      *bdir;
            struct bucket_desc      *bdesc, *prev;

            /* Calculate what page this object lives in */
            /* 计算该对象所在的页面 */
            page = (void *)  ((unsigned long) obj & 0xfffff000);
            /* Now search the buckets looking for that page */
            /* 现在搜索存储桶目录项所链接的桶描述符，寻找该页面 */
//
            for (bdir = bucket_dir; bdir->size; bdir++) {
                    prev = 0;
```

```
                /* If size is zero then this conditional is always false */
                /* 如果参数 size 是 0，则下面条件肯定是 false */
                if (bdir->size < size)
                        continue;
// 搜索对应目录项中链接的所有描述符，查找对应页面。如果某描述符页面指针等于 page 则表示找到
// 了相应的描述符，跳转到 found。如果描述符不含有对应 page，则让描述符指针 prev 指向该描述符。
                for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next) {
                        if (bdesc->page == page)
                                goto found;
                        prev = bdesc;
                }
        }
// 若搜索了对应目录项的所有描述符都没有找到指定的页面，则显示出错信息，死机。
        panic("Bad address passed to kernel free_s()");
found:
// 找到对应的桶描述符后，首先关中断。然后将该对象内存块链入空闲块对象链表中，并使该描述符
// 的对象引用计数减 1。
        cli(); /* To avoid race conditions */   /* 为了避免竞争条件 */
        *((void **)obj) = bdesc->freeptr;
        bdesc->freeptr = obj;
        bdesc->refcnt--;
// 如果引用计数已等于 0，则我们就可以释放对应的内存页面和该桶描述符。
        if (bdesc->refcnt == 0) {
                /*
                 * We need to make sure that prev is still accurate.  It
                 * may not be, if someone rudely interrupted us....
                 */
                /*
                 * 我们需要确信 prev 仍然是正确的，若某程序粗鲁地中断了我们
                 * 就有可能不是了。
                 */
// 如果 prev 已经不是搜索到的描述符的前一个描述符，则重新搜索当前描述符的前一个描述符。
                if ((prev && (prev->next != bdesc)) ||
                    (!prev && (bdir->chain != bdesc)))
                        for (prev = bdir->chain; prev; prev = prev->next)
                                if (prev->next == bdesc)
                                        break;
// 如果找到该前一个描述符，则从描述符链中删除当前描述符。
                if (prev)
                        prev->next = bdesc->next;
// 如果 prev==NULL，则说明当前一个描述符是该目录项首个描述符，也即目录项中 chain 应该直接
// 指向当前描述符 bdesc，否则表示链表有问题，则显示出错信息，死机。因此，为了将当前描述符
// 从链表中删除，应该让 chain 指向下一个描述符。
                else {
                        if (bdir->chain != bdesc)
                                panic("malloc bucket chains corrupted");
                        bdir->chain = bdesc->next;
                }
// 释放当前描述符所操作的内存页面，并将该描述符插入空闲描述符链表开始处。
                free_page((unsigned long) bdesc->page);
                bdesc->next = free_bucket_desc;
                free_bucket_desc = bdesc;
        }
```

```
        // 开中断，返回。
        sti();
        return;
}


```

12.10 open.c 程序

12.10.1 功能描述

open()系统调用用于将一个文件名转换成一个文件描述符。当调用成功时，返回的文件描述符将是进程没有打开的最小数值的描述符。该调用创建一个新的打开文件，并不与任何其它进程共享。在执行 exec 函数时，该新的文件描述符将始终保持着打开状态。文件的读写指针被设置在文件开始位置。

参数 flag 是 O_RDONLY、O_WRONLY、O_RDWR 之一，分别代表文件只读打开、只写打开和读写打开方式，可以与其它一些标志一起使用。(参见 fs/open.c，138 行)

12.10.2 代码注释

列表 linux/lib/open.c 程序

```
/*
 *  linux/lib/open.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define __LIBRARY__
#include <unistd.h>        // Linux 标准头文件。定义了各种符号常数和类型，并申明了各种函数。
                           // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()等。
#include <stdarg.h>        // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了-个
                           // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于
                           // vsprintf、vprintf、vfprintf 函数。

//// 打开文件函数。
// 打开并有可能创建一个文件。
// 参数：filename - 文件名；flag - 文件打开标志；...
// 返回：文件描述符，若出错则置出错码，并返回-1。
int open(const char * filename, int flag, ...)
{
        register int res;
        va_list arg;

// 利用 va_start()宏函数，取得 flag 后面参数的指针，然后调用系统中断 int 0x80，功能 open 进行
// 文件打开操作。
// %0 - eax(返回的描述符或出错码)；%1 - eax(系统中断调用功能号__NR_open)；
// %2 - ebx(文件名 filename)；%3 - ecx(打开文件标志 flag)；%4 - edx(后随参数文件属性 mode)。
```

```
        va_start(arg,flag);
        __asm__("int $0x80"
                :"=a" (res)
                :"" (__NR_open),"b" (filename),"c" (flag),
                "d" (va_arg(arg,int)));
// 系统中断调用返回值大于或等于 0，表示是一个文件描述符，则直接返回之。
        if (res>=0)
                return res;
// 否则说明返回值小于 0，则代表一个出错码。设置该出错码并返回-1。
        errno = -res;
        return -1;
}

```

12.11 setsid.c 程序

12.11.1 功能描述

该程序包括一个 setsid()系统调用函数。如果调用的进程不是一个组的领导时，该函数用于创建一个新会话。则调用进程将成为该新会话的领导、新进程组的组领导，并且没有控制终端。调用进程的组 id 和会话 id 被设置成进程的 PID(进程标识符)。调用进程将成为新进程组和新会话中的唯一进程。

12.11.2 代码注释

列表 linux/lib/setsid.c 程序

```
/*
 *  linux/lib/setsid.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define __LIBRARY__
#include <unistd.h>        // Linux 标准头文件。定义了各种符号常数和类型，并申明了各种函数。
                           // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()等。

//// 创建一个会话并设置进程组号。
// 下面系统调用宏对应于函数：pid_t setsid()。
// 返回：调用进程的会话标识符(session ID)。
_syscall0(pid_t,setsid)

```

12.12 string.c 程序

12.12.1 功能描述

所有字符串操作函数已经在 string.h 中实现，因此 string.c 程序仅包含 string.h 头文件。

12.12.2 代码注释

列标 linux/lib/string.c 程序

```
/*
 *  linux/lib/string.c
 *
 *  (C) 1991  Linus Torvalds
 */

#ifndef __GNUC__                // 需要 GNU 的 C 编译器编译。
#error I want gcc!
#endif

#define extern
#define inline
#define __LIBRARY__
#include <string.h>

```

12.13 wait.c 程序

12.13.1 功能描述

该程序包括函数 waitpid()和 wait()。这两个函数允许进程获取与其子进程之一的状态信息。各种选项允许获取已经终止或停止的子进程状态信息。如果存在两个或两个以上子进程的状态信息，则报告的顺序是不指定的。

wait()将挂起当前进程，直到其子进程之一退出（终止），或者收到要求终止该进程的信号，或者是需要调用一个信号句柄（信号处理程序）。

waitpid()挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止该进程的信号，或者是需要调用一个信号句柄（信号处理程序）。

如果 pid= -1，options=0，则 waitpid()的作用与 wait()函数一样。否则其行为将随 pid 和 options 参数的不同而不同。（参见 kernel/exit.c，142）

12.13.2 代码注释

列表 linux/lib/wait.c 程序

```
/*
 *  linux/lib/wait.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define __LIBRARY__
#include <unistd.h>         // Linux 标准头文件。定义了各种符号常数和类型，并申明了各种函数。
                            // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()等。
#include <sys/wait.h>       // 等待调用头文件。定义系统调用 wait()和 waitpid()及相关常数符号。

//// 等待进程终止系统调用函数。
// 该下面宏结构对应于函数：pid_t waitpid(pid_t pid, int * wait_stat, int options)
//
// 参数：pid - 等待被终止进程的进程 id，或者是用于指定特殊情况的其它特定数值；
//       wait_stat - 用于存放状态信息；options - WNOHANG 或 WUNTRACED 或是 0。
_syscall3(pid_t,waitpid,pid_t,pid,int *,wait_stat,int,options)

//// wait()系统调用。直接调用 waitpid()函数。
pid_t wait(int * wait_stat)
{
        return waitpid(-1,wait_stat,0);
}

```

12.14 write.c 程序

12.14.1 功能描述

该程序中包括一个向文件描述符写操作函数 write()。该函数向文件描述符指定的文件写入 count 字节的数据到缓冲区 buf 中。

12.14.2 代码注释

列表 linux/lib/write.c 程序

```
/*
 *  linux/lib/write.c
 *
 *  (C) 1991  Linus Torvalds
 */

#define __LIBRARY__
#include <unistd.h>         // Linux 标准头文件。定义了各种符号常数和类型，并申明了各种函数。
```

```
                                        // 如定义了__LIBRARY__，则还包括系统调用号和内嵌汇编_syscall0()
等。

//// 写文件系统调用函数。
// 该宏结构对应于函数：int write(int fd, const char * buf, off_t count)
// 参数：fd - 文件描述符；buf - 写缓冲区指针；count - 写字节数。
// 返回：成功时返回写入的字节数(0 表示写入 0 字节)；出错时将返回-1，并且设置了出错号。
_syscall3(int,write,int,fd,const char *,buf,off_t,count)

```

第 13 章 建造工具(tools)

13.1 概述

Linux 内核源代码中的 tools 目录中包含一个生成内核磁盘映象文件的工具程序 build.c，该程序将单独编译成可执行文件，在 linux/目录下的 Makefile 文件中被调用运行，用于将所有内核编译代码连接和合并成一个可运行的内核映像文件 image。具体方法是对 boot/中的 bootsect.s、setup.s 使用 8086 汇编器进行编译，分别生成各自的执行模块。再对源代码中的其它所有程序使用 GNU 的编译器 gcc/gas 进行编译，并连接成模块 system。然后使用 build 工具将这三块组合成一个内核映象文件 image。基本编译连接/组合结构如下图 13.1 所示。

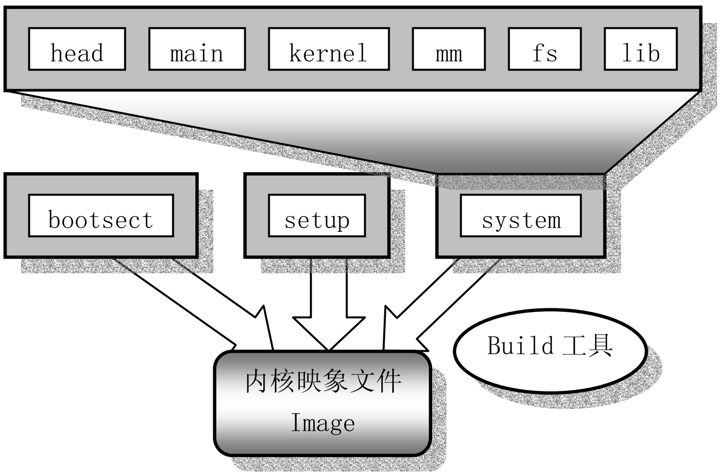


图 13.1 内核编译连接/组合结构

13.2 build.c 程序

13.2.1 功能概述

build 程序使用 4 个参数，分别是 bootsect 文件名、setup 文件名、system 文件名和可选的根文件系统设备文件名。

程序首先检查命令行上最后一个根设备文件名可选参数，若其存在，则读取该设备文件的状态信息结构（stat），取出设备号。若命令行上不带该参数，则使用默认值。

然后对 bootsect 文件进行处理，读取该文件的 minix 执行头部信息，判断其有效性，然后读取随后 512 字节的引导代码数据，判断其是否具有可引导标志 0xAA55，并将前面获取的根设备号写入到 508，509 位移处，最后将该 512 字节代码数据写到 stdout 标准输出，由 Make 文件重定向到 Image 文件。

接下来以类似的方法处理 setup 文件。若该文件长度小于 4 个扇区，则用 0 将其填满为 4 个扇区的长度，并写到标准输出 stdout 中。

最后处理 system 文件。该文件是使用 GCC 编译器产生，所以其执行头部格式是 GCC 类型的，与 linux 定义的 a.out 格式一样。在判断执行入口点是 0 后，就将数据写到标准输出 stdout 中。若其代码数据长度超过 128KB，则显示出错信息。

13.2.2 代码注释

列表 linux/tools/build.c 程序

```
/*
 *  linux/tools/build.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * This file builds a disk-image from three different files:
 *
 * - bootsect: max 510 bytes of 8086 machine code, loads the rest
 * - setup: max 4 sectors of 8086 machine code, sets up system parm
 * - system: 80386 code for actual system
 *
 * It does some checking that all files are of the correct type, and
 * just writes the result to stdout, removing headers and padding to
 * the right amount. It also writes some system data to stderr.
 */
/*
 * 该程序从三个不同的程序中创建磁盘映象文件：
 *
 * - bootsect：该文件的 8086 机器码最长为 510 字节，用于加载其它程序。
 * - setup：该文件的 8086 机器码最长为 4 个磁盘扇区，用于设置系统参数。
 * - system：实际系统的 80386 代码。
 *
 * 该程序首先检查所有程序模块的类型是否正确，并将检查结果在终端上显示出来，
 * 然后删除模块头部并扩充大正确的长度。该程序也会将一些系统数据写到 stderr。
 */

/*
 * Changes by tytso to allow root device specification
 */
/*
 * tytso 对该程序作了修改，以允许指定根文件设备。
 */

#include <stdio.h>       /* fprintf */            /* 使用其中的 fprintf() */
#include <string.h>                               /* 字符串操作 */
#include <stdlib.h>      /* contains exit */      /* 含有 exit() */
#include <sys/types.h>   /* unistd.h needs this */ /* 供 unistd.h 使用 */
#include <sys/stat.h>                             /* 文件状态信息结构 */
```

```
#include <linux/fs.h>                                 /* 文件系统 */
#include <unistd.h>       /* contains read/write */ /* 含有 read()/write() */
#include <fcntl.h>                                    /* 文件操作模式符号常数 */

#define MINIX_HEADER 32               // minix 二进制模块头部长度为 32 字节。
#define GCC_HEADER 1024               // GCC 头部信息长度为 1024 字节。

#define SYS_SIZE 0x2000               // system 文件最长节数(字节数为 SYS_SIZE*16=128KB)。

#define DEFAULT_MAJOR_ROOT 3          // 默认根设备主设备号 - 3(硬盘)。
#define DEFAULT_MINOR_ROOT 6          // 默认根设备次设备号 - 6(第 2 个硬盘的第 1 分区)。

/* max nr of sectors of setup: don't change unless you also change
 * bootsect etc */
/* 下面指定 setup 模块占的最大扇区数：不要改变该值，除非也改变 bootsect 等相应文件。
#define SETUP_SECTS 4                 // setup 最大长度为 4 个扇区(4*512 字节)。

#define STRINGIFY(x) #x               // 用于出错时显示语句中表示扇区数。

//// 显示出错信息，并终止程序。
void die(char * str)
{
        fprintf(stderr,"%s\n",str);
        exit(1);
}

// 显示程序使用方法，并退出。
void usage(void)
{
        die("Usage: build bootsect setup system [rootdev] [> image]");
}

int main(int argc, char ** argv)
{
        int i,c,id;
        char buf[1024];
        char major_root, minor_root;
        struct stat sb;

// 如果程序命令行参数不是 4 或 5 个，则显示程序用法并退出。
        if ((argc != 4) && (argc != 5))
                usage();
// 如果参数是 5 个，则说明带有根设备名。
        if (argc == 5) {
// 如果根设备名是软盘("FLOPPY")，则取该设备文件的状态信息，若出错则显示信息，退出。
                if (strcmp(argv[4], "FLOPPY")) {
                        if (stat(argv[4], &sb)) {
                                perror(argv[4]);
                                die("Couldn't stat root device.");
                        }
// 若成功则取该设备名状态结构中的主设备号和次设备号。
                        major_root = MAJOR(sb.st_rdev);
                        minor_root = MINOR(sb.st_rdev);
```

```
                } else {
// 否则让主设备号和次设备号取 0。
                        major_root = 0;
                        minor_root = 0;
                }
// 若参数只有 4 个，则让主设备号和次设备号等于系统默认的根设备。
        } else {
                major_root = DEFAULT_MAJOR_ROOT;
                minor_root = DEFAULT_MINOR_ROOT;
        }
// 在标准错误终端上显示所选择的根设备主、次设备号。
        fprintf(stderr, "Root device is (%d, %d)\n", major_root, minor_root);
// 如果主设备号不等于 2(软盘)或 3(硬盘)，也不等于 0(取系统默认根设备)，则显示出错信息，退出。
        if ((major_root != 2) && (major_root != 3) &&
            (major_root != 0)) {
                fprintf(stderr, "Illegal root device (major = %d)\n",
                        major_root);
                die("Bad root device --- major #");
        }
// 初始化 buf 缓冲区，全置 0。
        for (i=0;i<sizeof buf; i++) buf[i]=0;
// 以只读方式打开参数 1 指定的文件(bootsect)，若出错则显示出错信息，退出。
        if ((id=open(argv[1],O_RDONLY,0))<0)
                die("Unable to open 'boot'");
// 读取文件中的 minix 执行头部信息(参见列表后说明)，若出错则显示出错信息，退出。
        if (read(id,buf,MINIX_HEADER) != MINIX_HEADER)
                die("Unable to read header of 'boot'");
// 0x0301 - minix 头部 a_magic 魔数；0x10 - a_flag 可执行；0x04 - a_cpu, Intel 8086 机器码。
        if (((long *) buf)[0]!=0x04100301)
                die("Non-Minix header of 'boot'");
// 判断头部长度字段 a_hdrlen（字节）是否正确。（后三字节正好没有用，是 0）
        if (((long *) buf)[1]!=MINIX_HEADER)
                die("Non-Minix header of 'boot'");
// 判断数据段长 a_data 字段(long)内容是否为 0。
        if (((long *) buf)[3]!=0)
                die("Illegal data segment in 'boot'");
// 判断堆 a_bss 字段(long)内容是否为 0。
        if (((long *) buf)[4]!=0)
                die("Illegal bss in 'boot'");
// 判断执行点 a_entry 字段(long)内容是否为 0。
        if (((long *) buf)[5] != 0)
                die("Non-Minix header of 'boot'");
// 判断符号表长字段 a_sym 的内容是否为 0。
        if (((long *) buf)[7] != 0)
                die("Illegal symbol table in 'boot'");
// 读取实际代码数据，应该返回读取字节数为 512 字节。
        i=read(id,buf,sizeof buf);
        fprintf(stderr,"Boot sector %d bytes.\n",i);
        if (i != 512)
                die("Boot block must be exactly 512 bytes");
// 判断 boot 块 0x510 处是否有可引导标志 0xAA55。
        if ((*(unsigned short *)(buf+510)) != 0xAA55)
                die("Boot block hasn't got boot flag (0xAA55)");
```

```
// 引导块的 508，509 偏移处存放的是根设备号。
        buf[508] = (char) minor_root;
        buf[509] = (char) major_root;
// 将该 boot 块 512 字节的数据写到标准输出 stdout，若写出字节数不对，则显示出错信息，退出。
        i=write(1,buf,512);
        if (i!=512)
                die("Write call failed");
// 最后关闭 bootsect 模块文件。
        close (id);

// 现在开始处理 setup 模块。首先以只读方式打开该模块，若出错则显示出错信息，退出。
        if ((id=open(argv[2],O_RDONLY,0))<0)
                die("Unable to open 'setup'");
// 读取该文件中的 minix 执行头部信息(32 字节)，若出错则显示出错信息，退出。
        if (read(id,buf,MINIX_HEADER) != MINIX_HEADER)
                die("Unable to read header of 'setup'");
// 0x0301 - minix 头部 a_magic 魔数；0x10 - a_flag 可执行；0x04 - a_cpu, Intel 8086 机器码。
        if (((long *) buf)[0]!=0x04100301)
                die("Non-Minix header of 'setup'");
// 判断头部长度字段 a_hdrlen（字节）是否正确。（后三字节正好没有用，是 0）
        if (((long *) buf)[1]!=MINIX_HEADER)
                die("Non-Minix header of 'setup'");
// 判断数据段长 a_data 字段(long)内容是否为 0。
        if (((long *) buf)[3]!=0)
                die("Illegal data segment in 'setup'");
// 判断堆 a_bss 字段(long)内容是否为 0。
        if (((long *) buf)[4]!=0)
                die("Illegal bss in 'setup'");
// 判断执行点 a_entry 字段(long)内容是否为 0。
        if (((long *) buf)[5] != 0)
                die("Non-Minix header of 'setup'");
// 判断符号表长字段 a_sym 的内容是否为 0。
        if (((long *) buf)[7] != 0)
                die("Illegal symbol table in 'setup'");
// 读取随后的执行代码数据，并写到标准输出 stdout。
        for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
                if (write(1,buf,c)!=c)
                        die("Write call failed");
//关闭 setup 模块文件。
        close (id);
// 若 setup 模块长度大于 4 个扇区，则算出错，显示出错信息，退出。
        if (i > SETUP_SECTS*512)
                die("Setup exceeds " STRINGIFY(SETUP_SECTS)
                        " sectors - rewrite build/boot/setup");
// 在标准错误 stderr 显示 setup 文件的长度值。
        fprintf(stderr,"Setup is %d bytes.\n",i);
// 将缓冲区 buf 清零。
        for (c=0 ; c<sizeof(buf) ; c++)
                buf[c] = '\0';
// 若 setup 长度小于 4*512 字节，则用\0 将 setup 填足为 4*512 字节。
        while (i<SETUP_SECTS*512) {
                c = SETUP_SECTS*512-i;
                if (c > sizeof(buf))
```

```
                        c = sizeof(buf);
                if (write(1,buf,c) != c)
                        die("Write call failed");
                i += c;
        }

// 下面处理 system 模块。首先以只读方式打开该文件。
        if ((id=open(argv[3],O_RDONLY,0))<0)
                die("Unable to open 'system'");
// system 模块是 GCC 格式的文件，先读取 GCC 格式的头部结构信息(linux 的执行文件也采用该格式)。
        if (read(id,buf,GCC_HEADER) != GCC_HEADER)
                die("Unable to read header of 'system'");
// 该结构中的执行代码入口点字段 a_entry 值应为 0。
        if (((long *) buf)[5] != 0)
                die("Non-GCC header of 'system'");
// 读取随后的执行代码数据，并写到标准输出 stdout。
        for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
                if (write(1,buf,c)!=c)
                        die("Write call failed");
// 关闭 system 文件，并向 stderr 上打印 system 的字节数。
        close(id);
        fprintf(stderr,"System is %d bytes.\n",i);
// 若 system 代码数据长度超过 SYS_SIZE 节（或 128KB 字节），则显示出错信息，退出。
        if (i > SYS_SIZE*16)
                die("System is too big");
        return(0);
}
```

13.2.3 相关信息

可执行文件头部数据结构

Minix 可执行文件 a.out 的头部结构如下所示（参见 minix 2.0 源代码 01400 行开始）:

```
struct  exec {
  unsigned char a_magic[2];     // 执行文件魔数。
  unsigned char a_flags;        // 标志（参见后面说明）。
  unsigned char a_cpu;          // cpu 标识号。
  unsigned char a_hdrlen;       // 头部长度。
  unsigned char a_unused;       // 保留给将来使用。
  unsigned short a_version;     // 版本信息（目前未用）。
  long          a_text;         // 代码段长度，字节数。
  long          a_data;         // 数据段长度，字节数。
  long          a_bss;          // 堆长度，字节数。
  long          a_entry;        // 执行入口点地址。
  long          a_total;        // 分配的内存总量。
  long          a_syms;         // 符号表大小。
  ...
};
```

其中标志字段定义为：

```
A_UZP    0x01          // 未映射的 0 页（页数）。
A_PAL    0x02          // 以页边界调整的可执行文件。
A_NSYM   0x04          // 新型符号表。
A_EXEC   0x10          // 可执行文件。
A_SEP    0x20          // 代码和数据是分开的。
```

CPU 标识号为：

```
A_NONE   0x00          // 未知。
A_I8086 0x04           // Intel i8086/8088。
A_M68K   0x0B          // Motorola m68000。
A_NS16K 0x0C           // 国家半导体公司 16032。
A_I80386 0x10          // Intel i80386。
A_SPARC 0x17           // Sun 公司 SPARC。
```

GCC 执行文件头部结构信息参见 linux/include/a.out.h 文件。

参考文献

[1] Intel Co. INTEL 80386 Programmer's Reference Manual 1986, INTEL CORPORATION,1987.
[2] James L. Turley. Advanced 80386 Programming Technigues. Osborne McGraw-Hill,1988.
[3] Brian W. Kernighan, Dennis M. Ritchie. The C programming Language. Prentice-Hall 1988.
[4] Leland L. Beck. System Software: An Introduction to Systems Programming,3nd. Addison-Wesley,1997.
[5] Richard M. Stallman, Using and Porting the GNU Compiler Collection, For GCC Version 2.95, the Free Software Foundation, 1998.
[6] The Open Group Base Specifications Issue 6 IEEE Std 1003.1-2001, The IEEE and The Open Group.
[7] David A Rusling, The Linux Kernel, 1999. http://www.tldp.org/
[8] Linux Kernel Source Code，http://www.kernel.org/
[9] Digital co.ltd. VT100 User Guide, http://www.vt100.net/
[10] FreeBSD Online Manual, http://www.freebsd.org/cgi/man.cgi
[11] Maurice J. Bach 著，陈葆珏，王旭，柳纯录，冯雪山译，UNIX 操作系统设计. 机械工业出版社，2000 年 4 月.
[12] John Lions 著，尤晋元译，莱昂氏 UNIX 源代码分析，机械工业出版社，2000 年 7 月.
[13] Andrew S. Tanenbaum 著 王鹏，尤晋元等译，操作系统：设计与实现（第 2 版）, 电子工业出版社，1998 年 8 月.
[14] Alessandro Rubini，Jonathan 著，魏永明，骆刚，姜君译，Linux 设备驱动程序，中国电力出版社，2002 年 11 月.
[15] Daniel P. Bovet, Marco Cesati 著, 陈莉君，冯锐，牛欣源 译，深入理解 LINUX 内核, 中国电力出版社 2001 年.
[16] 张载鸿. 微型机(PC 系列)接口控制教程, 清华大学出版社，1992 年.
[17] 李凤华，周利华，赵丽松. MS-DOS 5.0 内核剖析. 西安电子科技大学出版社，1992 年.
[18] RedHat 7.3 操作系统在线手册. http://www.plinux.org/cgi-bin/man.cgi
[19] Ivan Bowman, Conceptual Architecture of the Linux Kernel. http://plg.uwaterloo.ca/~itbowman/CS746G/a1/
[20] Clark L. Coleman. Using Inline Assembly with gcc. http://oldlinux.org/Linux.old/
[21]

附录

附录 1 内核主要常数

1 系统最大进程数

系统最大进程（任务）数为 64。

2 进程的运行状态

```
#define TASK_RUNNING            0   // 进程正在运行或已准备就绪。
#define TASK_INTERRUPTIBLE      1   // 进程处于可中断等待状态。
#define TASK_UNINTERRUPTIBLE    2   // 进程处于不可中断等待状态，主要用于 I/O 操作等待。
#define TASK_ZOMBIE             3   // 进程处于僵死状态，已经停止运行，但父进程还没发信号。
#define TASK_STOPPED            4   // 进程已停止。
```

3 内存页长度

PAGE_SIZE = 1024 字节

3 系统主设备编号

与 Minix 系统的设备编号一样，因此可以使用 minix 的文件系统。

0 - 没有用到（nodev）
1 - /dev/mem 内存设备。
2 - /dev/fd 软盘设备。
3 - /dev/hd 硬盘设备。
4 - /dev/ttyx tty 串行终端设备。
5 - /dev/tty tty 终端设备。
6 - /dev/lp 打印设备。
7 - unnamed pipes 没有命名的管道。

4 硬盘逻辑设备编号方法

由于 1 个硬盘中可以存在 1--4 个分区，因此硬盘还依据分区的不同用次设备号进行指定分区。因此硬盘的逻辑设备号由以下方式构成：

设备号=主设备号*256 + 次设备号
也即 dev_no = (major<<8) + minor
两个硬盘的所有逻辑设备号见下表所示。

附表 1.1 硬盘逻辑设备号

逻辑设备号	对应设备文件	说明
0x300	/dev/hd0	代表整个第 1 个硬盘
0x301	/dev/hd1	表示第 1 个硬盘的第 1 个分区

0x302	/dev/hd2	表示第 1 个硬盘的第 2 个分区
0x303	/dev/hd3	表示第 1 个硬盘的第 3 个分区
0x304	/dev/hd4	表示第 1 个硬盘的第 4 个分区
0x305	/dev/hd5	代表整个第 2 个硬盘
0x306	/dev/hd6	表示第 2 个硬盘的第 1 个分区
0x307	/dev/hd7	表示第 2 个硬盘的第 2 个分区
0x308	/dev/hd8	表示第 2 个硬盘的第 3 个分区
0x309	/dev/hd9	表示第 2 个硬盘的第 4 个分区

其中 0x300 和 0x305 并不与哪个分区对应，而是代表整个硬盘。

从 linux 内核 0.95 版后已经不使用这种烦琐的命名方式，而是使用与现在相同的命名方法了。

附录 2 内核数据结构

这里集中列出了内核中的主要数据结构，并给予简单说明，注明了每个结构所在的文件和具体位置。作为阅读时参考。

1 执行文件结构 a.out（include/a.out.h，第 6 行）

a.out（Assembly out）执行文件头格式结构。

```
struct exec {
        unsigned long a_magic        // 执行文件魔数。使用 N_MAGIC 等宏访问。
        unsigned a_text              // 代码长度，字节数。
        unsigned a_data              // 数据长度，字节数。
        unsigned a_bss               // 文件中的未初始化数据区长度，字节数。
        unsigned a_syms              // 文件中的符号表长度，字节数。
        unsigned a_entry             // 执行开始地址。
        unsigned a_trsize            // 代码重定位信息长度，字节数。
        unsigned a_drsize            // 数据重定位信息长度，字节数。
};
```

2 文件锁定操作结构 flock（include/fcntl.h，43 行）

文件锁定操作数据结构。

```
struct flock {
        short l_type;          // 锁定类型（F_RDLCK，F_WRLCK，F_UNLCK）。
        short l_whence;        // 开始偏移(SEEK_SET，SEEK_CUR 或 SEEK_END)。
        off_t l_start;         // 阻塞锁定的开始处。相对偏移（字节数）。
        off_t l_len;           // 阻塞锁定的大小；如果是 0 则为到文件末尾。
        pid_t l_pid;           // 加锁的进程 id。
};
```

3 sigaction 的数据结构（include/signal.h，48 行）

sigaction 的数据结构。

```
struct sigaction {
        void (*sa_handler)(int);
        sigset_t sa_mask;
        int sa_flags;
        void (*sa_restorer)(void);
};
```

sa_handler 是对应某信号指定要采取的行动。可以是上面的 SIG_DFL，或者是 SIG_IGN 来忽略该信号，也可以是指向处理该信号函数的一个指针。

sa_mask 给出了对信号的屏蔽码，在信号程序执行时将阻塞对这些信号的处理。另外，引起触发信号处理的信号也将被阻塞，除非使用了 SA_NOMASK 标志。

sa_flags 指定改变信号处理过程的信号集。

sa_restorer 恢复过程指针，是用于保存原返回的过程指针。

4 终端窗口大小属性结构（include/termios.h，36 行）

窗口大小(Window size)属性结构。在窗口环境中可用于基于屏幕的应用程序。ioctls 中的 TIOCGWINSZ 和 TIOCSWINSZ 可用来读取或设置这些信息。

```
struct winsize {
        unsigned short ws_row;       // 窗口字符行数。
        unsigned short ws_col;       // 窗口字符列数。
        unsigned short ws_xpixel;    // 窗口宽度，象素值。
        unsigned short ws_ypixel;    // 窗口高度，象素值。
};
```

5 termio(s)结构（include/termios.h，44 行）

AT&T 系统 V 的 termio 结构。其中控制字符数据长度 NCC = 8。

```
struct termio {
        unsigned short c_iflag;        // 输入模式标志。
        unsigned short c_oflag;        // 输出模式标志。
        unsigned short c_cflag;        // 控制模式标志。
        unsigned short c_lflag;        // 本地模式标志。
        unsigned char c_line;          // 线路规程（速率）。
        unsigned char c_cc[NCC];       // 控制字符数组。
};
```

POSIX 的 termios 结构（第 54 行）。其中控制字符数据长度 NCC = 17。

```
struct termios {
        unsigned long c_iflag;         // 输入模式标志。
        unsigned long c_oflag;         // 输出模式标志。
        unsigned long c_cflag;         // 控制模式标志。
        unsigned long c_lflag;         // 本地模式标志。
        unsigned char c_line;          // 线路规程（速率）。
        unsigned char c_cc[NCCS];      // 控制字符数组。
};
```

以上定义的两个终端数据结构 termio 和 termios 是分别属于两类 UNIX 系列（或刻隆），termio 是在 AT&T 系统 V 中定义的，而 termios 是 POSIX 标准指定的。两个结构基本一样，只是 termio 使用短整数类型定义模式标志集，而 termios 使用长整数定义模式标志集。由于目前这两种结构都在使用，因此为了兼容性，大多数系统都同时支持它们。另外，以前使用的是一类似的 sgtty 结构，目前已基本不用。

6 时间结构（include/time.h，第 18 行）

```
struct tm {
        int tm_sec;                  // 秒数 [0，59]。
        int tm_min;                  // 分钟数 [ 0，59]。
        int tm_hour;                 // 小时数 [0，59]。
        int tm_mday;                 // 1 个月的天数 [0，31]。
        int tm_mon;                  // 1 年中月份 [0，11]。
        int tm_year;                 // 从 1900 年开始的年数。
        int tm_wday;                 // 1 星期中的某天 [0，6]（星期天 =0）。
```

```
        int tm_yday;                  // 1 年中的某天 [0，365]。
        int tm_isdst;                 // 夏令时标志。
};
```

7 文件访问/修改结构（include/utime.h，第 6 行）

```
struct utimbuf {
        time_t actime;            // 文件访问时间。从 1970.1.1:0:0:0 开始的秒数。
        time_t modtime;           // 文件修改时间。从 1970.1.1:0:0:0 开始的秒数。
};
```

8 缓冲区头结构 buffer_head（include/linux/fs.h，第 68 行）

缓冲区头数据结构。在程序中常用 bh 来表示 buffer_head 类型变量的缩写。

```
struct buffer_head {
        char * b_data;                   // 指向数据块的指针（数据块为 1024 字节）。
        unsigned long b_blocknr;        // 块号。
        unsigned short b_dev;           // 数据源的设备号（0 表示未用）。
        unsigned char b_uptodate;       // 更新标志：表示数据是否已更新。
        unsigned char b_dirt;           // 修改标志：0-未修改，1-已修改。
        unsigned char b_count;          // 使用该数据块的用户数。
        unsigned char b_lock;           // 缓冲区是否被锁定，0-未锁；1-已锁定。
        struct task_struct * b_wait;    // 指向等待该缓冲区解锁的任务。
        struct buffer_head * b_prev;    // 前一块（这四个指针用于缓冲区的管理）。
        struct buffer_head * b_next;    // 下一块。
        struct buffer_head * b_prev_free;   // 前一空闲块。
        struct buffer_head * b_next_free;   // 下一空闲块。
};
```

9 内存中磁盘索引节点结构（include/linux/fs.h，第 93 行）

这是在内存中的 i 节点结构。磁盘上的索引节点结构 d_inode 只包括前 7 项。

```
struct m_inode {
        unsigned short i_mode;        // 文件类型和属性(rwx 位)。
        unsigned short i_uid;         // 用户 id（文件拥有者标识符）。
        unsigned long i_size;         // 文件大小（字节数）。
        unsigned long i_mtime;        // 修改时间（自 1970.1.1:0 算起，秒）。
        unsigned char i_gid;          // 组 id(文件拥有者所在的组)。
        unsigned char i_nlinks;       // 文件目录项链接数。
        unsigned short i_zone[9];     // 直接(0-6)、间接(7)或双重间接(8)逻辑块号。
                                      // zone 是区的意思，可译成区段，或逻辑块。
/* these are in memory also */
        struct task_struct * i_wait;  // 等待该 i 节点的进程。
        unsigned long i_atime;        // 最后访问时间。
        unsigned long i_ctime;        // i 节点自身修改时间。
        unsigned short i_dev;         // i 节点所在的设备号。
        unsigned short i_num;         // i 节点号。
```

```
        unsigned short i_count;       // i 节点被使用的次数，0 表示该 i 节点空闲。
        unsigned char i_lock;         // 锁定标志。
        unsigned char i_dirt;         // 已修改(脏)标志。
        unsigned char i_pipe;         // 管道标志。
        unsigned char i_mount;        // 安装标志。
        unsigned char i_seek;         // 搜寻标志(lseek 时)。
        unsigned char i_update;       // 更新标志。
};
```

10 文件结构（include/linux/fs.h，第 116 行）

文件结构，用于在文件句柄与 i 节点之间建立关系。

```
struct file {
        unsigned short f_mode;        // 文件操作模式（RW 位）
        unsigned short f_flags;       // 文件打开和控制的标志。
        unsigned short f_count;       // 对应文件句柄（文件描述符）数。
        struct m_inode * f_inode;     // 指向对应 i 节点。
        off_t f_pos;                  // 文件位置（读写偏移值）。
};
```

11 磁盘超级块结构（include/linux/fs.h，第 124 行）

内存中磁盘超级块结构。磁盘上的超级块结构 d_super_block 只包括前 8 项。

```
struct super_block {
        unsigned short s_ninodes;       // 节点数。
        unsigned short s_nzones;        // 逻辑块数。
        unsigned short s_imap_blocks;   // i 节点位图所占用的数据块数。
        unsigned short s_zmap_blocks;   // 逻辑块位图所占用的数据块数。
        unsigned short s_firstdatazone; // 第一个数据逻辑块号。
        unsigned short s_log_zone_size; // log(数据块数/逻辑块)。(以 2 为底)。
        unsigned long s_max_size;       // 文件最大长度。
        unsigned short s_magic;         // 文件系统魔数。
/* These are only in memory */
        struct buffer_head * s_imap[8]; // i 节点位图缓冲块指针数组(占用 8 块，可表示 64M)。
        struct buffer_head * s_zmap[8]; // 逻辑块位图缓冲块指针数组（占用 8 块）。
        unsigned short s_dev;           // 超级块所在的设备号。
        struct m_inode * s_isup;        // 被安装的文件系统根目录的 i 节点。(isup-super i)
        struct m_inode * s_imount;      // 被安装到的 i 节点。
        unsigned long s_time;           // 修改时间。
        struct task_struct * s_wait;    // 等待该超级块的进程。
        unsigned char s_lock;           // 被锁定标志。
        unsigned char s_rd_only;        // 只读标志。
        unsigned char s_dirt;           // 已修改(脏)标志。
};
```

12 目录项结构（include/linux/fs.h，第 157 行）

文件目录项结构。

```
struct dir_entry {
        unsigned short inode;         // i 节点。
        char name[NAME_LEN];          // 文件名。
};
```

13 硬盘分区表结构（include/linux/hdreg.h，第 52 行）

硬盘分区表结构。参见下面列表后信息。

```
struct partition {
        unsigned char boot_ind;       // 引导标志。0x80-该分区可引导操作系统。
        unsigned char head;           // 分区起始磁头号。
        unsigned char sector;         // 分区起始扇区号(位 0-5)和起始柱面号高 2 位(位 6-7)。
        unsigned char cyl;            // 分区起始柱面号低 8 位。
        unsigned char sys_ind;       // 分区类型字节。0x0b-DOS; 0x80-Old Minix; 0x83-Linux
        unsigned char end_head;       // 分区的结束磁头号。
        unsigned char end_sector;     // 结束扇区号(位 0-5)和结束柱面号高 2 位(位 6-7)。
        unsigned char end_cyl;        // 结束柱面号低 8 位。
        unsigned int start_sect;      // 分区起始物理扇区号（从 0 开始计)。
        unsigned int nr_sects;        // 分区占用的扇区数。
};
```

为了实现多个操作系统共享硬盘资源，硬盘可以在逻辑上分为 1--4 个分区。每个分区之间的扇区号是邻接的。分区表由 4 个表项组成，每个表项由 16 字节组成，对应一个分区的信息，存放有分区的大小和起止的柱面号、磁道号、扇区号和引导标志。分区表存放在硬盘的 0 柱面 0 头第 1 个扇区的 0x1BE--0x1FD 处。4 个分区中同时只能有一个分区是可引导的。

14 段描述符结构（include/linux/head.h，第 4 行）

CPU 中描述符的简单格式。

```
struct desc_struct {                 // 定义了段描述符的数据结构。该结构仅说明每个描述
        unsigned long a,b;           // 符是由 8 个字节构成，每个描述符表共有 256 项。
} desc_table[256];
```

15 i387 使用的结构（include/linux/sched.h，第 40 行）

这是数学协处理器使用的结构，主要用于保存进程切换时 i387 的执行状态信息。

```
struct i387_struct {
        long    cwd;        // 控制字(Control word)。
        long    swd;        // 状态字(Status word)。
        long    twd;        // 标记字(Tag word)。
        long    fip;        // 协处理器代码指针。
        long    fcs;        // 协处理器代码段寄存器。
        long    foo;
        long    fos;
        long    st_space[20];   /* 8*10 bytes for each FP-reg = 80 bytes */
```

```
};
```

16 任务状态段结构（include/linux/sched.h，第 51 行）

任务状态段数据结构（参见附录）。

```
struct tss_struct {
        long    back_link;      /* 16 high bits zero */
        long    esp0;
        long    ss0;            /* 16 high bits zero */
        long    esp1;
        long    ss1;            /* 16 high bits zero */
        long    esp2;
        long    ss2;            /* 16 high bits zero */
        long    cr3;
        long    eip;
        long    eflags;
        long    eax,ecx,edx,ebx;
        long    esp;
        long    ebp;
        long    esi;
        long    edi;
        long    es;             /* 16 high bits zero */
        long    cs;             /* 16 high bits zero */
        long    ss;             /* 16 high bits zero */
        long    ds;             /* 16 high bits zero */
        long    fs;             /* 16 high bits zero */
        long    gs;             /* 16 high bits zero */
        long    ldt;            /* 16 high bits zero */
        long    trace_bitmap;   /* bits: trace 0, bitmap 16-31 */
        struct i387_struct i387;
};
```

17 进程（任务）数据结构 task（include/linux/sched.h，第 78 行）

这是任务（进程）数据结构，或称为进程描述符。

struct task_struct {

    long state   任务的运行状态（-1 不可运行，0 可运行(就绪)，>0 已停止）。
    long counter   任务运行时间计数(递减)（滴答数），运行时间片。
    long priority   运行优先数。任务开始运行时 counter = priority，越大运行越
长。
    long signal   信号。是位图，每个比特位代表一种信号，信号值=位偏移值+1。
    struct sigaction sigaction[32] 信号执行属性结构，对应信号将要执行的操作和标志信息。
    long blocked   进程信号屏蔽码（对应信号位图）。
    int exit_code   任务执行停止的退出码，其父进程会取。
    unsigned long start_code   代码段地址。
    unsigned long end_code   代码长度（字节数）。

```
    unsigned long end_data      数据长度（字节数）。
    unsigned long brk           总长度（字节数）。
    unsigned long start_stack 堆栈段地址。
    long pid                    进程标识号(进程号)。
    long father                 父进程号。
    long pgrp                   父进程组号。
    long session                会话号。
    long leader                 会话首领。
    unsigned short uid          用户标识号（用户 id）。
    unsigned short euid         有效用户 id。
    unsigned short suid         保存的用户 id。
    unsigned short gid          组标识号（组 id）。
    unsigned short egid         有效组 id。
    unsigned short sgid         保存的组 id。
    long alarm                  报警定时值（滴答数）。
    long utime                  用户态运行时间（滴答数）。
    long stime                  系统态运行时间（滴答数）。
    long cutime                 子进程用户态运行时间。
    long cstime                 子进程系统态运行时间。
    long start_time             进程开始运行时刻。
    unsigned short used_math    标志：是否使用了协处理器。
    int tty                     进程使用 tty 的子设备号。-1 表示没有使用。
    unsigned short umask        文件创建属性屏蔽位。
    struct m_inode * pwd        当前工作目录 i 节点结构。
    struct m_inode * root       根目录 i 节点结构。
    struct m_inode * executable 执行文件 i 节点结构。
    unsigned long close_on_exec 执行时关闭文件句柄位图标志。（参见 include/fcntl.h）
    struct file * filp[NR_OPEN] 进程使用的文件表结构。
    struct desc_struct ldt[3]   任务的局部描述符表。0-空，1-代码段 cs，2-数据和堆栈段 ds&ss。
    struct tss_struct tss       进程的任务状态段信息结构。
};
```

18 tty 等待队列结构（include/linux/tty.h，第 16 行）

tty 等待队列数据结构。

```
struct tty_queue {
        unsigned long data;             // 等待队列缓冲区中当前数据指针（字符数[??]）。
                                        // 对于串口终端，则存放串口端口地址。
        unsigned long head;             // 缓冲区中数据头指针。
        unsigned long tail;             // 缓冲区中数据尾指针。
        struct task_struct * proc_list; // 等待进程列表。
        char buf[TTY_BUF_SIZE];         // 队列的缓冲区。
};
```

19 tty 结构（include/linux/tty.h，第 45 行）

tty 数据结构。

```
struct tty_struct {
        struct termios termios;                     // 终端 io 属性和控制字符数据结构。
        int pgrp;                                   // 所属进程组。
        int stopped;                                // 停止标志。
        void (*write)(struct tty_struct * tty);     // tty 写函数指针。
        struct tty_queue read_q;                    // tty 读队列。
        struct tty_queue write_q;                   // tty 写队列。
        struct tty_queue secondary;                 // tty 辅助队列(存放规范模式字符序列),
};                                                  // 可称为规范(熟)模式队列。
extern struct tty_struct tty_table[];               // tty 结构数组。
```

20 文件状态结构（include/sys/stat.h，第 6 行）

```
struct stat {
        dev_t    st_dev;     // 含有文件的设备号。
        ino_t    st_ino;     // 文件 i 节点号。
        umode_t  st_mode;    // 文件属性（见下面）。
        nlink_t  st_nlink;   // 指定文件的连接数。
        uid_t    st_uid;     // 文件的用户(标识)号。
        gid_t    st_gid;     // 文件的组号。
        dev_t    st_rdev;    // 设备号(如果文件是特殊的字符文件或块文件)。
        off_t    st_size;    // 文件大小（字节数）(如果文件是常规文件)。
        time_t   st_atime;   // 上次（最后）访问时间。
        time_t   st_mtime;   // 最后修改时间。
        time_t   st_ctime;   // 最后节点修改时间。
};
```

21 文件访问与修改时间结构（include/sys/times.h，第 6 行）

```
struct tms {
        time_t tms_utime;   // 用户使用的 CPU 时间。
        time_t tms_stime;   // 系统（内核）CPU 时间。
        time_t tms_cutime;  // 已终止的子进程使用的用户 CPU 时间。
        time_t tms_cstime;  // 已终止的子进程使用的系统 CPU 时间。
};
```

22 ustat 结构（include/sys/types.h，第 39 行）

```
struct ustat {
        daddr_t f_tfree;
        ino_t f_tinode;
        char f_fname[6];
        char f_fpack[6];
};
```

23 系统名称头文件（include/sys/utsname.h，第 6 行）

```
struct utsname {
        char sysname[9];   // 本版本操作系统的名称。
        char nodename[9];  // 与实现相关的网络中节点名称。
        char release[9];   // 本实现的当前发行级别。
        char version[9];   // 本次发行的版本级别。
        char machine[9];   // 系统运行的硬件类型名称。
};
```

24 块设备请求项结构（kernel/blk_dev/blk.h，第 23 行）

下面是请求队列中项的结构。其中如果 dev=-1，则表示没有使用该项。

```
struct request {
        int dev;                          // 使用的设备号，未用时为-1。
        int cmd;                          // 命令(READ 或 WRITE)。
        int errors;                       // 作时产生的错误次数。
        unsigned long sector;             // 起始扇区。(1 块=2 扇区)
        unsigned long nr_sectors;         // 读/写扇区数。
        char * buffer;                    // 数据缓冲区。
        struct task_struct * waiting;     // 任务等待操作执行完成的地方。
        struct buffer_head * bh;          // 缓冲区头指针(include/linux/fs.h,68)。
        struct request * next;            // 指向下一请求项。
};
```

附录 3 80x86 保护运行模式

80386 概述

80386 是一个高级的 32 位微处理器，专门用于多任务的操作系统，并为需要高性能的应用所设计。32 位的寄存器和数据通道支持 32 位的寻址方式和数据类型，处理器可以寻址最高可达 4GB 的物理内存以及 64TB（2^{46}字节）的虚拟内存。芯片上的内存管理包括地址转换寄存器、高级多任务硬件、保护机制以及分页虚拟内存机制。下面针对系统编程，概要说明使用 80386 的这些基本原理。

系统寄存器

设计用于系统编程的系统寄存器主要包括以下几类：

标志寄存器 EFALGS；
内存管理寄存器；
控制寄存器；
调试寄存器；
测试寄存器。

系统标志寄存器 EFLAGS 控制着 I/O、可屏蔽中断、调试、任务切换以及保护模式和多任务环境下虚拟 8086 程序的执行。其中主要标志见下图所示。

31								23								15								7							0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	VM	RF	0	NT	IO PL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

其中系统标志：VM – 虚拟 8086 模式；RF – 恢复标志；NT – 嵌套任务标志；IO PL – I/O 特权级标志；IF – 中断允许标志。

内存管理寄存器有 4 个，用于分段内存管理：

- GDTR – 全局描述符表寄存器(Global Descriptor Table Register)；
- LDTR – 局部描述符表寄存器(Local Descriptor Table Register)；
- IDTR – 中断描述符表寄存器(Interrupt Descriptor Table Register)；
- TR – 任务寄存器。

其中前两个寄存器(GDTR,LDTR)分别指向段描述符表 GDT 和 LDT。IDTR 寄存器指向中断向量表。TR 寄存器指向处理器所需的当前任务的信息。

80386 共有 4 个控制寄存器，分别是 CR0、CR1、CR2 和 CR3。格式见下图所示。

31	23	15	7 0	
页目录基地址寄存器 Page Directory Base Register (PDBR)			保留 Reserved	CR3
页异常线性地址 Page Fault Linear Address				CR2
保留 Reserved				CR1

P	保留	E	T	E	M	P	CR0
G	Reserved	T	S	M	P	E	

控制寄存器 CR0 含有系统整体的控制标志。其中：

- PE – 保护模式开启位（Protection Enable，比特位 0）。如果设置了该比特位，就会使处理器开始在保护模式下运行。
- MP – 协处理器存在标志（Math Present，比特位 1）。用于控制 WAIT 指令的功能，以配合协处理的运行。
- EM – 仿真控制（Emulation，比特位 2）。指示是否需要仿真协处理器的功能。
- TS – 任务切换（Task Switch，比特位 3）。每当任务切换时处理器就会设置该比特位，并且在解释协处理器指令之前测试该位。
- ET – 扩展类型（Extention Type，比特位 4）。该位指出了系统中所含有的协处理器类型（是 80287 还是 80387）。
- PG – 分页操作（Paging，比特位 31）。该位指示出是否使用页表将线性地址变换成物理地址。

内存管理

内存管理主要涉及处理器的内存寻址机制。80x86 使用两步将一个分段形式的逻辑地址转换为实际物理内存地址。

- 段变换，将一个由段选择符和段内偏移构成的逻辑地址转换为一个线性地址；
- 页变换，将线性地址转换为对应的物理地址。该步是可选的。

在分页机制开启时，通过将前面所述的段转换和页转换组合在一起，即实现了从逻辑地址到物理地址的两个转换阶段。

段变换

下图示出了处理器是如何将一个逻辑地址转换为线性地址的。在转换过程中 CPU 使用了以下一些数据结构：

- 段描述符（Segment Descriptors）；
- 描述符表（Descriptor tables）；
- 选择符（Selectors）；
- 段寄存器（Segment Registers）。

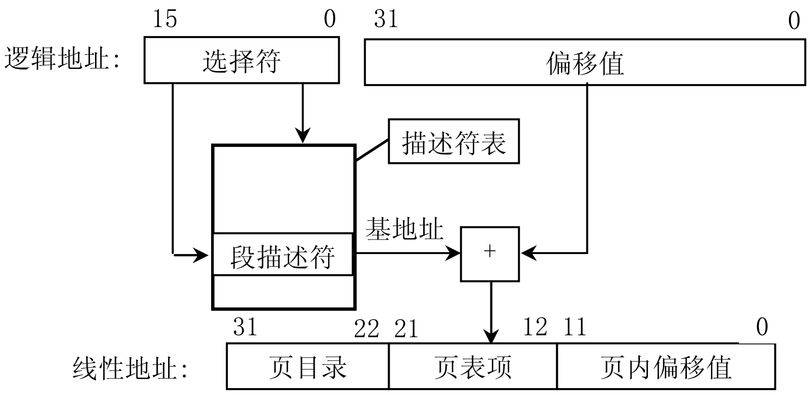


图 段变换示意图

段描述符

段描述符向 CPU 提供了将逻辑地址映射为线性地址所必要的信息。描述符是由程序编译器、链接器、加载器或操作系统创建的。下图示出了描述符的两种一般格式。所有种类的描述符都具有这两种格式之一。段描述符的各个字段的含义如下：

<table>
<tr><td>31</td><td colspan="5">23</td><td colspan="5">15</td><td>7   0</td><td></td></tr>
<tr><td>基地址(BASE)
位 31..24</td><td>G</td><td>X</td><td>0</td><td>A
V
L</td><td>限长(LIMIT)
位 19..16</td><td>P</td><td>DPL</td><td>1</td><td>TYPE</td><td>A</td><td>基地址(BASE)
位 23..16</td><td>4</td></tr>
<tr><td colspan="6">段基地址(BASE) 位 15..0</td><td colspan="6">段限长(LIMIT) 位 15..0</td><td>0</td></tr>
</table>

a.用于程序代码段和数据段的描述符

<table>
<tr><td>31</td><td colspan="5">23</td><td colspan="4">15</td><td>7   0</td><td></td></tr>
<tr><td>基地址(BASE)
位 31..24</td><td>G</td><td>X</td><td>0</td><td>A
V
L</td><td>限长(LIMIT)
位 19..16</td><td>P</td><td>DPL</td><td>0</td><td>TYPE</td><td>基地址(BASE)
位 23..16</td><td>4</td></tr>
<tr><td colspan="6">段基地址(BASE) 位 15..0</td><td colspan="5">段限长(LIMIT) 位 15..0</td><td>0</td></tr>
</table>

b.用于特殊系统段的描述符

图 描述符的一般格式

基地址（BASE）：定义段在 4GB 线性空间中的位置。处理器会将基地址的三个部分组合成一个 32 位的值。
段限长（LIMIT）：定义了段的最大长度。处理器将组合段限长的两个部分形成一个 20 位的值。处理器会依据颗粒度（Granularity）位字段的值来解释段限长域的实际含义：

1. 当以 1 字节为单元时，则定义了最高可为 1MB 字节的长度；
2. 当以 4KB 字节为单元时，则定义了最高可为 4GB 字节的长度。在加载时限长值将左移 12 位。

颗粒度（Granularity）：指定了限长字段值代表的单元含义。当为 0 时，限长单元值为 1 字节；当该位为 1 时，限长的单元值为 4KB 字节。
类型（TYPE）：用于区分各种不同类型的描述符。
描述符特权级（Descriptor Privilege Level – DPL）：用于保护机制。共有 4 级：0－3。0 级是最高特权级，3 级是最低特权级。
段存在位（Segment-Present bit – P）：如果该位为零，则该描述符无效，不能用于地址变换过程。当指向该描述符的选择符被加载到段寄存器中时，处理器就会发出一个异常信号。
访问位（Accessed bit – A）：当处理器访问过该段时就会设置该比特位。

描述符表

段描述符是保存在描述符表中的，有两类描述符表：

- 全局描述符表（Global descriptor table – GDT）；
- 局部描述符表（Local descriptor table – LDT）。

描述符表是由 8 字节构成的描述符项的内存中的一个数组，见下图所示。描述符表的长度是可变的，最多可以含有 8192（2^{13}）个描述符。但是对于 GDT 表，其第一个描述符（索引 0）是不用的。

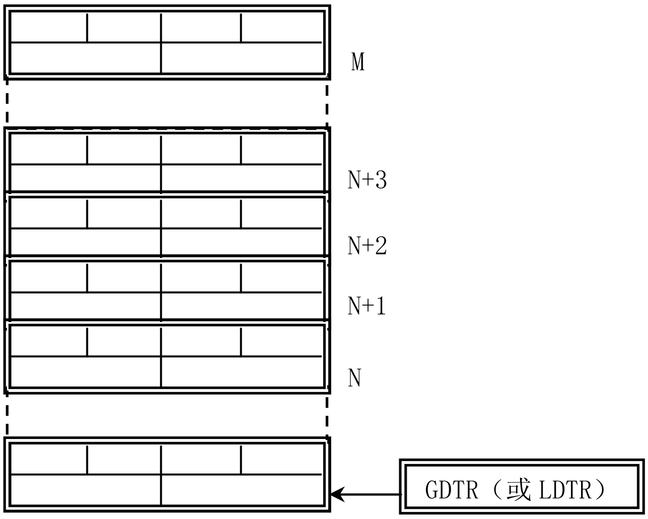


图 描述符表示意图

处理器是通过使用 GDTR 和 LDTR 寄存器来定位 GDT 表和当前的 LDT 表。这两个寄存器以线性地址的方式保存了描述符表的基地址和表的长度。指令 lgdt 和 sgdt 用于访问 GDTR 寄存器；指令 lldt 和 sldt 用于访问 LDTR 寄存器。lgdt 使用的是内存中一个 6 字节操作数来加载 GDTR 寄存器的。头两个字节代表描述符表的长度，后 4 个字节是描述符表的基地址。然而请注意，访问 LDTR 寄存器的指令 lldt 所使用的操作数却是一个 2 字节的操作数，表示全局描述符表 GDT 中一个描述符项的选择符。该选择符所对应的 GDT 表中的描述符项应该对应一个局部描述符表。选择符的含义见下面说明。

31	23	15	7　　　　0	
基地址（BASE）				2
		表限长（LIMIT）		0

选择符(Selectors)

逻辑地址的选择符部分是用于指定一描述符的，它是通过指定一描述符表并且索引其中的一个描述符项完成的。下图示出了选择符的格式。各字段的含义为：

索引值(Index)：用于选择指定描述符表中 8192 个描述符中的一个。处理器将该索引值乘上 8(描述符的字节长度)，并加上描述符表的基地址即可访问表中指定的段描述符。

表指示器(Table Indicator - TI)：指定选择符所引用的描述符表。值为 0 表示指定 GDT 表，值为 1 表示指定当前的 LDT 表。

请求者的特权级(Requestor's Privalege Level - RPL)：用于保护机制。

15　　　　3	2	1　0
索引值 (INDEX)	T I	RPL

由于 GDT 表的第一项(索引值为 0)没有被使用，因此一个具有索引值 0 和表指示器值也为 0 的选择符（也即指向 GDT 的第一项的选择符）可以用作为一个空(null)选择符。当一个段寄存器（不能是 CS 或 SS）加载了一个空选择符时，处理器并不会产生一个异常。但是若使用这个段寄存器访问内存时就会

产生一个异常。对于初始化还未使用的段寄存器以陷入意外的引用来说，这个特性是很有用的。

段寄存器

处理器将描述符中的信息保存在段寄存器中，因而可以避免在每次访问内存时查询描述符表。

每个段寄存器都有一个“可见”部分和一个“不可见”部分，见下图所示。这些段地址寄存器的可见部分是由程序来操作的，就好象它们只是简单的16位寄存器。不可见部分则是由处理器来处理的。

对这些寄存器的加载操作使用的是普通程序指令，这些指令可以分为两类：

1. 直接加载指令；例如，MOV，POP，LDS，LSS，LGS，LFS。这些指令显式地引用了指定的段寄存器。
2. 隐式加载指令；例如，远调用CALL和远跳转JMP。这些指令隐式地引用了CS段寄存器，并用新值加载到CS中。

程序使用这些指令会把16位的选择符加载到段寄存器的可见部分，而处理器则会自动地从描述符表中将一个描述符的基地址、段限长、类型以及其它信息加载到段寄存器中的不可见部分中去。

	16位可见部分	隐藏部分
CS		
SS		
DS		
ES		
FS		
GS		

页变换（翻译）

在地址变换的第二阶段，CPU将线性地址转换为物理地址。地址变换的这个阶段实现了基于分页的虚拟内存系统和分页级保护的基本功能。页变化这一步是可选的，页变换仅在设置了CR0的PG比特位后才起作用，该比特位是在软件初始化时由操作系统设置的。如果操作系统需要实现多个虚拟8086任务、基于分页的保护机制或基于分页的虚拟内存，那么就一定要设置该位。

页框（帧）(Page Frame)

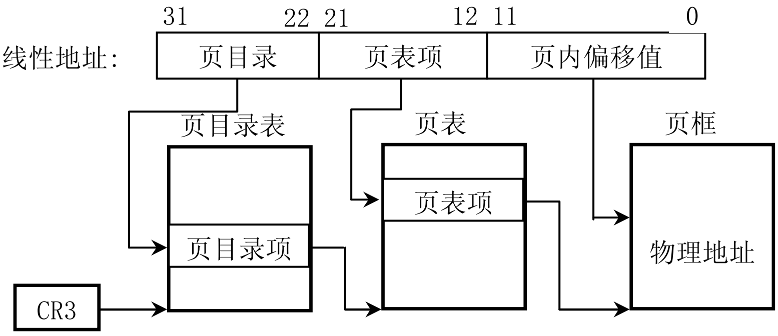
页框是一个物理内存地址连续的4K字节单元。它以字节为边界，大小固定。

线性地址(Linear Address)

线性地址通过指定一个页表、页表中的某一页以及该页中的偏移值，从而间接地指向对应的物理地址。下图示出了线性地址的格式。

31　　　22	21　　　12	11　　　0
页目录 (DIR)	页 (PAGE)	偏移值 (OFFSET)

下图示出了处理器将一个线性地址转换成物理地址的方法。通过使用两级页表，处理器将一个线性地址的页目录字段(DIR)、页字段(PAGE)和偏移字段(OFFSET)翻译成对应的物理地址。寻址机制使用线性地址的页目录字段作为页目录中的索引值、使用页表字段作为页目录所指定页表中的索引值、使用偏移字段作为页表所确定的内存页中的字节偏移值。



页表(Page Table)

页表只是一个简单的 32 位页指示器的数组。页表本身也是一页内存，因此它含有 4K 字节的内存，可容纳 1K 个 32 位的项。

这里使用了两级页表来定位一页内存页。最高层是页目录，页目录可定位最多 1K 个第二级页表，而每个二级页表可以定位最多 1K 内存页。因此，一个页目录定位的所有页表可以寻址 1M 内存页（2^{20}）。由于每一页内存含有 4K 字节（2^{12}），最终一个页目录所指定的页表可以寻址 80386 的整个物理地址空间（$2^{20} * 2^{12} = 2^{32}$）。

当前页目录的物理地址是存储在 CPU 控制寄存器 CR3 中的，因此该寄存器也被称为页目录基地址寄存器（page directory base register – PDBR）。内存管理软件可以选择对所有的任务只使用一个页目录，或每个任务使用一个页目录，也可以组合两个任务使用一个页目录。

页表项(Page-Table Entries)

各级页表所使用的页表项是相同的，其格式见下图所示。

31 12	11									0
页框地址 位 31..12 (PAGE FRAME ADDRESS)	可用 (AVAIL)	0	0	D	A	0	0	U/S	R/W	P

其中，页框地址(PAGE FRAME ADDRESS)指定了一页内存的物理起始地址。因为内存页是位于 4K 边界上的，所以其低 12 比特总是 0。在一个页目录中，页表项的页框地址是一个页表的起始地址；在第二级页表中，页表项的页框地址是包含期望内存操作的页框的地址。

存在位（PRESENT – P）确定了一个页表项是否可以用于地址转换过程。P=1 表示该项可用。当目录表项或第二级表项的 P=0 时，则该表项时无效的，不能用于地址转换过程。此时该表项的其它所有比特位都可供程序使用；处理器不对这些位进行测试。

当 CPU 试图使用一个页表项进行地址转换时，如果此时任意一级页表项的 P=0，则处理器就会发出页异常信号。对于支持分页虚拟内存的软件系统中，页不存在(page-not-present)异常处理程序就可以把所请求的页加入到物理内存中。此时导致异常的指令就可以被重新执行。

已访问（Accessed – A）和已修改（Dirty – D）比特位提供了有关页使用的信息。除了页目录项中的已修改位，这些比特位将由硬件置位，但不复位。

在对一页内存进行读或写操作之前，处理器将设置相关的目录和二级页表项的已访问位。在向一个二级页表项所涵盖的地址进行写操作之前，处理器将设置该二级页表项的已修改位，而页目录项中的已修改位是不用的。当需求的内存超出实际物理内存量时，支持分页虚拟内存的操作系统可以使用这些位

来确定那些页可以从内存中取走。操作系统必须负责检测和复位这些比特位。

读/写位（Read/Write – R/W）和用户/超级用户位（User/Supervisor – U/S）并不用于地址转换，但用于分页级的保护机制，是由处理器在地址转换过程中同时操作的。

页转换高速缓冲

为了最大地提高地址转换的效率，处理器将最近所使用的页表数据存放在芯片上的高速缓冲中。操作系统设计人员必须在当前页表改变时刷新高速缓冲，可使用以下两种方式之一：

1. 通过使用 MOV 指令重新加载 CR3 页目录基址寄存器；
2. 通过执行一个任务切换。

多任务（Multitasking）

为了提供有效的、受保护的多任务机制，80x86 使用了一些特殊的数据结构。支持多任务运行的寄存器和数据结构主要有任务状态段（Task State Segment）和任务寄存器（Task register）。使用这些数据结构，CPU 可以快速地从一个任务的执行切换到另一个任务，并保存原有任务的内容。

任务状态段（Task State Segment – TSS）

处理器管理一个任务的所有信息存储在一个特殊类型的段中，即任务状态段 TSS。下图给出了 TSS 的格式。其中的字段可分为两类：

- 处理器只读其中信息的静态字段集（图中灰色部分）；
- 每次任务切换时处理器将会更新的动态字段集。

31　　23	15　　7　　0	
I/O 映射图基地址(MAP BASE)	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	64
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	局部描述符表(LDT)的选择符	60
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	GS	5C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	FS	58
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DS	54
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS	50
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	CS	4C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ES	48
EDI		44
ESI		40
EBP		3C
ESP		38
EBX		34
EDX		30
ECX		2C
EAX		28
EFLAGS		24
指令指针(EIP)		20
页目录基地址寄存器 CR3（PDBR）		1C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS2	18
ESP2		14
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS1	10

ESP1		0C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS0	08
ESP0		04
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	前一执行任务 TSS 的描述符	00

任务状态段 TSS 可以处于线性空间的任何位置。TSS 与其它段一样，也是使用段描述符来定义的。访问 TSS 的描述符会导致任务切换。因此，在大多数系统中都将描述符的 DPL（描述符特权级）字段设置为最高特权级 0，这样就可以只允许可信任的软件执行任务的切换。TSS 的描述符只能放在全局描述符表 GDT 中。

任务寄存器

任务寄存器（Task Register - TR）的作用与一般段寄存器的类似，它通过指向 TSS 来确定当前执行的任务。它也有 16 位的可见部分和不可见部分。可见部分中的选择符用于在 GDT 表中选择一个 TSS 描述符，处理器使用不可见部分来存放描述符中的基地址和段限长值。指令 LTR 和 STR 用于修改和读取任务寄存器中的可见部分，指令所使用的操作数是一 16 位的选择符。

另外，还有一种提供对 TSS 间接、受保护引用的任务门描述符（Task Gete Descriptor）。这种描述符是在一般段描述符格式的基地址位 15..0 字段(第 3、4 字节)中存放的是一个 TSS 描述符的选择符，并利用其中的特权级字段(DPL)来控制使用描述符执行任务切换的权限。见下面有关中断描述符表 IDT 描述符中的说明。

在以下 4 种情况下，CPU 会切换执行的任务：

1. 当前任务执行了一条引用 TSS 描述符的 JMP 或 CALL 指令；
2. 当前任务执行了一条引用任务门的 JMP 或 CALL 指令；
3. 引用了中断描述符表（IDT）中任务门的中断或异常；
4. 当嵌套任务标志 NT 置位时，当前任务执行了一个 IRET 指令。

中断和异常

中断和异常是一种特殊类型的控制转换。它们改变了正常程序流而去处理其它的事件（例如外部事件、出错报告或异常条件）。中断与异常的主要区别在于中断常用于处理 CPU 外部的异步事件，而异常则是处理 CPU 在执行过程中本身检测到的问题。

外部中断源有两种：由 CPU 的 INTR 引脚输入的可屏蔽中断和 NMI 引脚输入的不可屏蔽中断。同样，异常也有两类：由 CPU 检测到的出错、陷阱或放弃事件以及编程设置的“软中断”（如 INT 3 指令等)。

处理器使用标识号（中断号）来识别每种类型的中断或异常。处理器所能识别的不可屏蔽中断 NMI 和异常的标识号是预先确定的，范围是 0 到 31（0x00-0x1f）。目前这些号码并没有全都使用，未确定的号码由 Intel 公司留作今后使用。

可屏蔽中断的标识号由外部中断控制器（如 8259A 可编程中断控制器）确定，并在 CPU 的中断识别阶段通知 CPU。8259A 所分配的中断号可以通过编程指定，可使用的标识号范围是 32 到 255（0x20-0xff）。Linux 系统将 32-47 分配给了可屏蔽中断，余下的 48-255 用来标识其它软中断。当 Linux 只使用了号码 128（0x80）作为系统调用的中断向量号。

中断描述符表

中断描述符表（Interrupt Descriptor Table - IDT）将每个中断或异常标识号与处理相应事件程序指令的一个描述符相关联。与 GDT 和 LDT 相似，IDT 是一个 8 字节描述符数组，但其第 1 项可以含有一个描述符。处理器通过将中断号异常号乘上 8 即可索引 IDT 中对应的描述符。IDT 可以位于物理内存的任何地方。处理器是使用 IDT 寄存器（IDTR）来定位 IDT 的。修改和复制 IDT 的指令是 LIDT 和 SIDT。

与 GDT 表的操作一样，IDT 也是使用 6 字节数据的内存地址作为操作数的。前两个字节表示表的限长，后 4 个字节是表的线性基地址。

IDT 描述符

在中断描述符表 IDT 中可以含有三类描述符中的任意一种：

- 任务门（Task gates）；
- 中断门（Interrupt gates）；
- 陷阱门（Trap gates）；

下图给出了任务门、中断门和陷阱门描述符的格式。

31 23	15			7 0	
（未使用）	P	DPL	0 0 1 0 1	（未使用）	4
TSS 段选择符（SELECTOR）	（未使用）				0

80X86 任务门描述符

31 23	15			7	0	
偏移值(OFFSET) 位 31..16	P	DPL	0 1 1 1 0	0 0 0	（未使用）	4
段选择符（SELECTOR）	偏移值（OFFSET）位 15..0					0

80X86 中断门描述符

31 23	15			7	0	
偏移值(OFFSET) 位 31..16	P	DPL	0 1 1 1 1	0 0 0	（未使用）	4
段选择符（SELECTOR）	偏移值（OFFSET）位 15..0					0

80X86 陷阱门描述符

中断任务和中断过程

正如 CALL 指令能调用一个过程或任务一样，一个中断或异常也能“调用”中断处理程序，该程序是一个过程或一个任务。当响应一个中断或异常时，CPU 使用中断或异常的标识号来索引 IDT 表中的描述符。如果 CPU 索引到一个中断门或陷阱门时，它就调用处理过程；如果是一个任务门，它就引起任务切换。

中断门或陷阱门间接地指向一个过程，该过程将在当前执行任务上下文中执行。门描述符中的段选择符指向 GDT 或当前 LDT 中的一个可执行段的描述符。门描述符中的偏移字段值指向中断或异常处理过程的开始处。

80X86 执行一个中断或异常处理过程的方式与 CALL 指令调用一个过程的方式非常相似，只是两者在使用堆栈上略有不同。中断会在把原指令指针压入堆栈之前，把原标志寄存器 EFLAGS 的内容也推入堆栈中。对于与段有关的异常，CPU 还会将一个错误码压入异常处理程序的堆栈上。

对于中断过程处理结束的返回操作，中断返回指令 IRET 与 RET 相似，但是 IRET 为了去除压入堆栈的 EFLAGS 值，ESP 会多递增 4 个字节。

中断门与陷阱门的区别在于对中断允许标志 IF 的影响。由中断门向量引起的中断会复位 IF，因为可以避免其它中断干扰当前中断的处理。随后的 IRET 指令会从堆栈上恢复 IF 的原值；而通过陷阱门产生的中断不会改变 IF。

IDT 表中的任务门描述符间接地指向一个任务状态段 TSS。任务门描述符中的段选择符指向 GDT 表中的一个 TSS 描述符。当产生的中断或异常指向 IDT 中的一个任务门描述符，就会导致任务切换，从而会在独立的任务中处理中断。Linux 系统中并没有使用任务门描述符。

索引

由于内核代码相对比较庞大，很多变量/函数在源代码的很多程序中被使用/调用，因此对其进行索引比较困难。本索引主要根据变量或函数名称给出定义它的程序文件名、行号和所在页码。